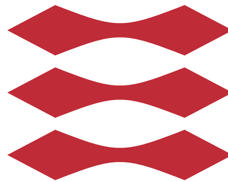
Presburger Arithmetic and its use in verification

Phan Anh Dung



Kongens Lyngby 2011

IMM-MSC-2011-39

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC-2011-39

Summary

Today, when every computer has gone multicore, the requirement of taking advantage of these computing powers becomes critical. However, multicore parallelism is complex and error-prone due to extensive use of synchronization techniques. In this thesis, we explore the functional paradigm in the context of F# programming language and parallelism support in .NET framework. This paradigm prefers the declarative way of thinking and no side effect which lead to the ease of parallelizing algorithms. The purpose is to investigate decision algorithms for quantified linear arithmetic of integers (also called Presburger Arithmetic) aiming at efficient parallel implementations. The context of the project is; however, to support model checking for the real-time logic Duration Calculus, and the goal is to decide a subset of Presburger formulas, which is relevant to this model-checking problem and for which efficient tool support can be provided. We present a simplification process for this subset of Presburger formulas which gives some hope for creating small corresponding formulas. Later two parallel decision procedures for Presburger Arithmetic along with their experimental results are discussed.

Preface

The thesis is a part of the double-degree Erasmus Mundus Mater in Security and Mobile Computing (NordSecMob), which consisted of study at the Royal Institute of Technology (KTH) and the Technical University of Denmark (DTU). This thesis was done at the department of Informatics and Mathematical Modelling (IMM) at DTU, under the main supervision of Associate Professor Michael R. Hansen from DTU and the co-supervision of Professor Mads Dam from KTH.

The thesis deals with various problems of Presburger Arithmetic ranging from simplification to decision procedures. The reader is expected to be familiar with F#, .NET framework and parallel programming. No prior knowledge of functional programming is needed, although it helps to understand the idea of multicore parallelism in connection with the functional paradigm.

This thesis was written during the period from January 2011 to June 2011.

Kongens Lyngby, June 2011

Phan Anh Dung

Acknowledgements

I would like to thank my supervisor Michael R. Hansen for his help on various problems emerging in this project. I really appreciate his encouragements whenever I have trouble; his valuable support helped me to finish my thesis on time. I would also like to thank Aske W. Brekling for fruitful discussions and his help on technical problems.

I would like to acknowledge the help from my co-supervisor Mads Dam; his tolerance and guidance is important to me to complete this work. Finally, I would like to thank my family and my girlfriend because of their support and understanding although I am far way from home.

List of Abbreviations

CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DC	Duration Calculus
DNF	Disjunctive Normal Form
GC	Garbage Collector
GNF	Guarded Normal Form
ITL	Interval Temporal Logic
LINQ	Language Integrated Query
NNF	Negation Normal Form
PA	Presburger Arithmetic
PFX	Parallel Extensions
PNF	Prenex Normal Form
PRAM	Parallel Random Access Machine
RAM	Random Access Machine
SAT	Satisfiability
TPL	Task Parallel Library
WSN	Wireless Sensor Network

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Presburger Arithmetic and problems of parallel decision procedures 3
- 1.3 Purpose of the thesis . . . 4

2 Multicore parallelism on F# and .NET framework **5**
- 2.1 Multicore parallelism: a brief overview . . . 5
- 2.2 Multicore parallelism on .NET framework . . . 10
- 2.3 Summary . . . 21

3 Functional paradigm and multicore parallelism **23**
- 3.1 Parallel functional algorithms . . . 23
- 3.2 Some pitfalls of functional parallelism on the multicore architecture 26
- 3.3 Summary . . . 31

4 Theory of Presburger Arithmetic **33**
- 4.1 Overview . . . 33
- 4.2 Decision Procedures for Presburger Arithmetic . . . 35
- 4.3 Summary . . . 41

5 Duration Calculus and Presburger fragments **43**
- 5.1 Duration Calculus and side-condition Presburger formulas . . . 43
- 5.2 Simplification of Presburger formulas . . . 47
- 5.3 Summary . . . 51

6 Experiments on Presburger Arithmetic **53**
- 6.1 Generation of Presburger fragments . . . 53
- 6.2 Simplification of Presburger formulas . . . 56
- 6.3 Optimization of Cooper's algorithm . . . 56

6.4 Summary . . . 58

7 Parallel execution of decision procedures **59**
7.1 A parallel version of Cooper's algorithm . . . 60
7.2 A parallel version of the Omega Test . . . 67
7.3 Summary . . . 72

8 Conclusions **73**

References **75**

A Examples of multicore parallelism in F# **79**
A.1 Source code of π calculation . . . 79
A.2 Source code of MergeSort . . . 82

B Source code of experiments **85**
B.1 Utilities.fs . . . 85
B.2 Term.fs . . . 87
B.3 Formula.fs . . . 90
B.4 PAGenerator.fs (excerpt) . . . 91
B.5 Cooper.fs (excerpt) . . . 93
B.6 OmegaTest.fs . . . 101
B.7 OmegaTestArray.fs . . . 103

CHAPTER 1

Introduction

1.1 Background

In 1965, Gordon Moore proposed Moore's Law predicting the number of transistors on an integrated circuit would double every 18-24 months resulting in a corresponding increase of processing power in the same time period [32]. Software used to get free extra speed whenever hardware manufacturers released newer and faster mainstream systems. However, when clock frequency of a single processor reached their peak a few years ago, the free lunch was almost over. Due to power consumption problem, instead of making more powerful and faster processors, CPU makers tried to integrate many cores into a single processor. In theory, CPU power still grows according to Moore's Law, but single-threaded applications cannot benefit from extra CPU resources anymore. Multicore trend helps chip designers to avoid serious problems such as heat losses and leakage current but still achieve good processing power at an economical price. However, multicore computing brings in a new challenge for software architects and developers, they need to reconsider the way which systems are built to be able to have good utilization of CPU resources.

Although parallel computing is a hinder for software development because of requiring so much time and effort for development, it brings significant opportunities for organizations being able to exploit parallelism. Organizations will

have new experiences of fast service delivery and efficient products leading to potential business opportunities. Some areas obviously requiring parallel computing are product design simulation where manufacturers are able to quickly prototype virtual products and financial modeling thanks to which financial companies can offer customers powerful modeling tools with rich analysis of financial scenarios. Also parallel programming is expected to brings benefits to numerical software products where heavy computation of numerical and symbolic algorithms requires efficient use of computing power.

Procedural and object-oriented programs are difficult to parallelize due to the problem of shared states and imperative data structures. This problem does not occur in functional programming, in this thesis we shall investigate how to exploit functional paradigm in the context of parallel programming. Functional programming has its clear advantages of supporting parallel computing. First, functional programming relies on data immutability which guarantees code execution without side effects; therefore, different parts of algorithms could be parallelized without introducing any synchronization construct. Second, the declarative way of programming enables developers to describe what problems are rather than how to solve them and consequently make them easier to break up and parallelize. Third, functional constructs such as high-order functions and lambda expressions provide convenient tools for clearly structuring the code, which eases the pain of prototyping parallel programs. F# is chosen as the functional programming language for development. Beside other advantages of a functional programming language, its well-supported .NET framework provides rich libraries for developing applications and efficient constructs for parallelism.

Later we review the idiom of functional paradigm and parallel execution along with decision procedures for Presburger Arithmetic (PA). These algorithms are difficult case studies of tool support; Presburger formulas are known to be decidable but their decision procedures are doubly exponential lower bound and triply exponential upper bound [23]. However, instances of PA keep appearing in compiler optimization and model checking problems, which raises the need for practically fast implementation of PA decision procedure. Some Presburger fragments are being used in connection with a model checker for Duration Calculus (DC) [10]. For example, power usage of nodes on a Wireless Sensor Network (WSN) is expressed in DC and later converted into a Presburger fragment. To be able to deduce conclusions about power usage, the Presburger formula which may appear to have rather big size has to be decided. Therefore, we perform experiments with parallelism and PA decision procedures using F# and .NET framework. Hopefully, these experiments can help us to get closer to the goal of efficient tool support for PA.

1.2 Presburger Arithmetic and problems of parallel decision procedures

Decision procedures for PA exist but they are quite expensive for practical usage [28]. There are various attempts to optimize those decision procedures in many aspects. However, those efforts only help to reduce memory usage and provide fast response for a certain type of formulas; no attempt on employing extra CPU power for PA algorithms is found in the academia. Although lack of reference for related work on the problem brings us a new challenge, we enlarge the investigation to parallel execution of decision procedures in general; hopefully understanding of their approaches might be helpful. As it turns out, parallelization of SAT solvers is a rather unexplored topic. Two main approaches are mainly used for parallel SAT solving. The first one is Search Space Splitting where search space is broken into independent parts and subproblems are solved in parallel. Typically in this approach, if one thread completes its work early, it will be assigned other tasks by a dynamic work-stealing mechanism. One example of this approach is the multithreaded ySAT by Feldman *et al.* [7]. The second approach is Algorithm Portfolio where the same instance is solved in parallel by different SAT solvers with different parameter settings. Learning process is important for achieving good efficiency in this method. ManySAT by Hamadi *et al.* [13] is known as a good example in this category.

Beside the problem of few related literature, parallel execution of PA is thought to be difficult due to many reasons:

- Algorithms designed in the sequential way of thinking need careful reviews, parallel execution of sequential algorithms will not bring benefits.
- Some algorithms are inherently difficult to parallelize. The degree of parallelism depends on how much work is run sequentially, if sequential part of the program dominates, parallelism does not help much.
- Achieving parallelism is much dependent on support of the programming language; moreover, parallelization is subtle and requires thorough profiling and testing process.

This thesis relies on investigation of decision procedures and parallel patterns. From this investigation, we are going to sketch some ideas for parallelism in PA decision procedures in connection with DC model checker.

1.3 Purpose of the thesis

In this work, we expect to achieve following objectives:

- Investigate combination between functional programming and parallel programming in the context of F# and .NET platform.
- Apply learned principles into problems inspired by studying decision procedures for PA.
- Investigate efficient algorithms for deciding Presburger fragments, particularly focus on Presburger formulas useful for Duration Calculus's Model Checker.
- Design and implement these decision procedures with concentration on parallel execution.

Chapter 2

Multicore parallelism on F# and .NET framework

The chapter starts with a brief overview of multicore parallelism. Although multicore parallelism is a special case of parallel processing which performs on the multicore architecture, the architecture actually has a huge influence on how multicore parallelism is done and what we can achieve. Later we introduce parallelism constructs by a series of familiar examples, we shall see that the functional paradigm is suitable and easily used to write correct parallel implementations.

2.1 Multicore parallelism: a brief overview

The section describes some useful concepts which will be used throughout the report. Although these concepts are terms of parallel processing, they should be understood in the specific context of multicore parallelism. We also present the architecture of multicore computers because it is essential and has its influence on what we can achieve in multicore parallelism.

2.1.1 Platform

Parallel computing is usually classified by platforms where they are executed. A platform could be a cluster of nodes where coordination is done by message passing or a single machine with multiple cores coordinating in parallel using shared memory. In this work, parallel processing is done on a multicore platform, so shared-memory parallel programming is used by default here. One thing needed to clarify is we focus on parallelism rather than concurrency. By parallelism, we mean to exploit all system resources to speedup computation as much as possible, so performance is the most important goal. This is different from concurrency where different jobs may be executed at the same time and may not block each other. Moreover, multicore is rather new compared to other parallel platforms so results on those platform require careful justification. The structure of multicore computers which brings some advantages and disadvantages to parallel computing is going to be discussed in the next section.

2.1.2 Architecture of multicore computers

Figure 2.1 illustrates a simplified structure of multicore machines. There are multiple cores with independent ALUs and instruction streams which enable computation to be done in parallel. While each core has their own L1 cache, L2 cache and main memory are shared between all cores. If data fit into L1 cache, all cores would operate in parallel. However, in practice some cores have to read memory blocks from L2 cache to L1 cache, or from main memory to L2 cache. Therefore, sequential access to L2 cache and main memory is a significant bottleneck in parallel execution. In general, memory access pattern and cache behaviour are important factors which affect performance of multicore parallel programs.

2.1.3 Parallel programming frameworks

One problem of parallel computing is the divergence of programming frameworks which makes developers confused and it is hard to choose a correct environment for their purpose. However, two most popular parallel programming frameworks are MPI for message passing and OpenMP for shared memory [21]. MPI consists of a set of library functions for process creation, message passing and communication operations. Its model assumes distributed memory so that MPI only fits distributed applications which work on a cluster of machines. OpenMP contains a set of compiler directives to add parallelism to existing code. Its model

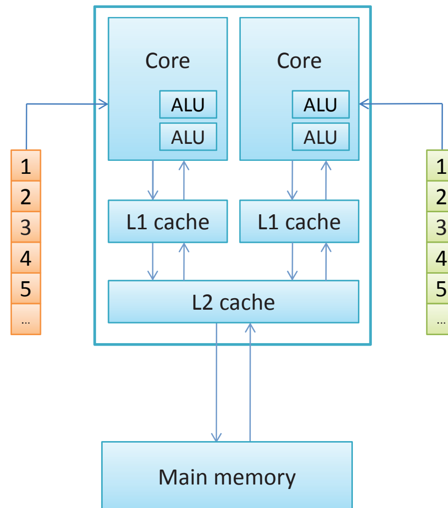


Figure 2.1: A common architecture of multicore hardware [29].

assumes the architecture of symmetric multiprocessors, thus working well on shared memory machines. Although OpenMP is easier to adapt to the architecture of multicore hardwares than MPI, it was not designed specifically for the multicore platform. Other new parallel frameworks including Parallel Extensions (PFX) of .NET framework are expected to exploit multicore computing powers better.

2.1.4 Programming Models

Categorized by programming models, parallel processing falls into two groups: **explicit parallelism** and **implicit parallelism** [12]. In explicit parallelism, concurrency is expressed by means of special-purpose directives or function calls. These primitives are related to synchronization, communication and task creation which result in parallel overheads. Explicit parallelism is criticized for being too complicated and hiding meanings of parallel algorithms; however, it provides programmers with full control over parallel execution, which leads to optimal performance. MPI framework and Java's parallelism constructs are well-known examples of explicit parallelism.

Implicit parallelism is based on compiler support to exploit parallelism without using additional constructs. Normally, implicit parallel languages do not require

special directives or routines to enable parallel execution. Some examples of implicit parallel languages could be HPF and NESL [12]; these languages hide complexity of task management and communication, so developers can focus on designing parallel programs. However, implicit parallelism does not give programmers much space for tweaking programs leading to suboptimal parallel efficiency.

In this work, we use F# on .NET framework for parallel processing. F#'s parallelism constructs are clearly explicit parallelism; in Section 2.2 we are showing that these constructs are very close to implicit parallelism but they still allow users to control the degree of parallelism.

2.1.5 Important Concepts

Speedup

Speedup is defined as follows [21]:

$$S_N = \frac{T_1}{T_N}$$

where:

- N is the number of processors.
- T_1 is the execution time of a sequential algorithm.
- T_N is the execution time of a parallel algorithm using N processors.

In an ideal case when $S_N = N$, a linear speedup is obtained. However, super linear speedup higher than N with N processors might happen. One reason might be cache effect, a larger cache size can make data fit into caches; therefore, memory access is faster and extra speedup is obtained not from actual computation. Another reason might be cooperation between tasks in a parallel backtracking algorithm. When performing backtracking in parallel, some branches of a search tree are pruned much earlier by some tasks, which results in a smaller search space for each task.

Parallel Efficiency

Parallel efficiency is defined by the following equation [21]:

$$\mathtt{E_N} = \frac{\mathtt{S_N}}{\mathtt{N}}$$

It is a performance metric, typically between zero and one, indicating CPU utilization of solving problems in parallel.

Amdahl's Law

Suppose that P is the portion of a program which can be parallelized and $1 - P$ is the portion which still remains sequential, the maximum speedup of using N processors is [16]:

$$\mathtt{S_N} = \frac{1}{(1 - \mathtt{P}) + \frac{\mathtt{P}}{\mathtt{N}}}$$

The optimal speedup tends to be $1/(1 - \mathtt{P})$ no matter how many processors are added. For example, if $P = 90\%$ performance could be speded up until a factor of 10. Amdahl's Law is criticized for being pessimistic in the assumption that the problem size is fixed. In some applications where data is highly independent from each other, when the problem size increases, the optimal speedup factor becomes rather big.

Gustafson's Law

Addressing the pitfall of Amdahl's Law with fixed problem sizes, Gustafson's Law states that the maximum speedup factor is [11]:

$$\mathtt{S_N} = \mathtt{N} - \alpha.(\mathtt{N} - 1)$$

where:

- N is the number of processors.
- α is the non-parallelizable portion of a program.

Gustafson's Law is more optimistic than Amdahl's; it implies that size of problems solved in a fixed time increases when computing power grows. Also it spots the need of minimizing the sequential portion of a program, even if this minimization increases total amount of computations.

Synchronization

Parallel processing allows two or more processes running simultaneously. In various cases, the order of events does not matter. In other cases, to ensure correctness of a program, we have to ensure that events occur in a specific order. Synchronization constructs are introduced to enforce constraints in these cases.

Race condition

This is a typical kind of error happening to parallel programs. It occurs when a program runs in the same system with the same data but produces totally different results. One cause of race condition is synchronization. Read and write commands to shared variables have to be done in a correct order; otherwise, results are unpredictably wrong.

Deadlocks

Deadlocks usually occur with parallel programs when complex coordination between tasks is employed. When a cycle of tasks are blocked and waiting for each other, a deadlock occurs. In general, deadlocks are not difficult to detect statically and some constructs are made for resolving deadlocks when they happen.

Parallel slowdown

When a task is divided into more and more subtasks, these subtasks spend more and more time communicating with each other; eventually, overheads of communication dominate running time, and further parallelization increases rather than decreases total running time. Therefore, good parallel efficiency requires careful management of task creation and task partitioning.

2.2 Multicore parallelism on .NET framework

In this section, we shall introduce parallelism constructs of .NET platform. To make the introduction more interesting, we demonstrate some specific examples

which directly use those parallelism constructs. All benchmarks here are done on an 8-core 2.40GHz Intel Xeon workstation with 8GB shared physical memory.

2.2.1 Overview

Parallel programming has been supported by .NET framework for quite some time, and it becomes really mature with Parallel Extension (PFX) in .NET 4.0. An overall picture of parallel constructs is illustrated in Figure 2.2. In general, a parallel program is written in an imperative way or in a functional way and compiled into any language in .NET platform. Here we are particularly interested in F# and how to use parallel constructs in F#. Some programs may employ PLINQ execution engine to run LINQ queries in parallel; however, behind the scene, PLINQ is based on Task Parallel Library (TPL) to enable parallelism. An important primitive in TPL which represents an execution unit is `Task`; a parallel program is divided into many `Tasks` which are able to execute in parallel. TPL runs tasks in parallel by the following procedure: it spawns a number of worker threads; tasks are put into a *Work Queue* which is accessible by all threads; each thread consumes a number of tasks and returns results. This *Work Queue* supports a work-stealing mechanism in which each thread can steal tasks of other threads when it finishes its work, and the number of worker threads is determined by TPL according to how heavy the tasks are. As we can see, TPL hides complexity of creating and managing threads which easily go wrong; therefore, users can focus on designing parallel algorithms rather than worry about low-level primitives. Anyway, users still have their capabilities of controlling execution of `Tasks` and the number of worker threads, which will be presented in next sections.

2.2.2 Data parallelism

Data parallelism is used when we have a collection of data and a single operation to apply on each item. Data parallelism clearly benefits us if items are independent of each other and cost of gathering data is not so big. The example demonstrated here is estimating the value of π. This value could be calculated by an integral as follows:

$$\pi = \int_0^1 \frac{4}{1 + x^2}\, dx$$

For purpose of demonstration, we divide the range from 0 to 1 into 100000000 steps and estimate the area (which is corresponding to π) by a trapezoid integration rule. The first version is using LINQ, a data query component in .NET

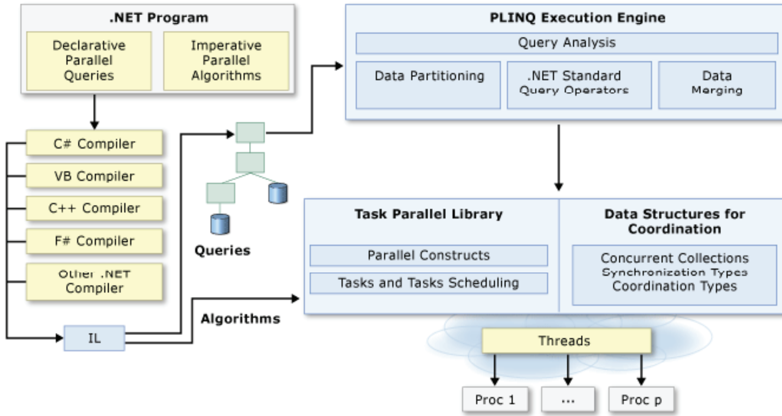


Figure 2.2: Parallel programming in .NET framework [29].

platform. LINQ allows users to write queries in a very declarative style which does not expose any side effect and shows nature of problems very clearly:

```
let NUM_STEPS = 100000000
let steps = 1.0 / (float NUM_STEPS)
let sqr x = x * x

let linqCompute1() =
    (Enumerable
      .Range(0, NUM_STEPS)
      .Select(fun i -> 4.0 / (1.0 + sqr((float i + 0.5) * steps)))
      .Sum()
      ) * steps
```

Computation is done in three steps. First, the range of 100,000,000 elements are constructed as an instance of `Enumerable` class, and this type is a central element for accommodating LINQ queries. Second, the `Select` method calculates intermediate values for all indices. Finally, the `Sum` function aggregates all immediate results. Due to the advantage of side effect free, LINQ queries are easy to parallelize. In reality, .NET platform has supported LINQ queries in a parallel manner by PLINQ constructs. The calculation π is turned into parallel execution by just changing `Enumerable` to `ParallelEnumerable` to indicate usage of PLINQ:

```
let plinqCompute2() =
    (ParallelEnumerable
      .Range(0, NUM_STEPS)
```

```
    .Select(fun i -> 4.0 / (1.0 + sqr((float i + 0.5) * steps)))
    .Sum()
    ) * steps
```

PLINQ provides users a cheap way to do parallelization in .NET framework; our PLINQ version is 3-4× faster than the LINQ companion. One question raised is: why not provide PLINQ as a default option for data querying in .NET. The reason is that PLINQ is not always faster than its corresponding LINQ companion. To be worth using PLINQ, a task should be big and contain rather independent subtasks. Certainly, parallel processing by using PLINQ has drawbacks of little control over degree of parallelism and the way tasks are split and distributed over processing units.

Our third version is a simple **for** loop running 100,000,000 times and gradually gathering results implemented by means of recursion in F#:

```
let compute3() =
    let rec computeUtil(i, acc) =
        if i = 0 then acc * steps
        else
            let x = (float i + 0.5) * steps
            computeUtil (i-1, acc + 4.0 / (1.0 + x * x))
    computeUtil(NUM_STEPS, 0.0)
```

Its parallel equivalent is written using **Parallel.For** construct:

```
let parallelCompute4() =
    let sum = ref 0.0
    let monitor = new Object()
    Parallel.For(
        0, NUM_STEPS, new ParallelOptions(),
        (fun () -> 0.0),
        (fun i loopState (local:float) ->
            let x = (float i + 0.5) * steps
            local + 4.0 / (1.0 + x * x)
            ),
        (fun local -> lock (monitor) (fun () -> sum := !sum + local))) |>
             ignore
    !sum * steps
```

This variant of **Parallel.For** loop consists of 6 parameters: the first two parameters are the first and the last array index; the third one specifies some advanced options to control parallelism; the fourth argument is to initialize local variables of worker threads; the fifth one is a computing function at each index and the last argument is the aggregated function to sum up all intermediate results to the final one. This version is 3-4× faster than the sequential

companion; however, we can improve more here. As we can see, the **Select** phase is fully parallel, but the **Sum** phase is partly sequential. Here we use **lock** to avoid concurrent writes to **sum** variable; acquiring lock is expensive so their use should be minimized as much as possible. We come out with a new parallel implementation as follows:

```
let parallelCompute5() =
    let rangeSize = NUM_STEPS / (Environment.ProcessorCount * 10)
    let partitions = Partitioner.Create(0, NUM_STEPS, if rangeSize >= 1 then
         rangeSize else 1)
    let sum = ref 0.0
    let monitor = new Object()
    Parallel.ForEach(
        partitions, new ParallelOptions(),
        (fun () -> 0.0),
        (fun (min, max) loopState l ->
            let local = ref 0.0
            for i in min .. max - 1 do
                let x = (float i + 0.5) * steps
                local := !local + 4.0 / (1.0 + x * x)
            l + !local),
        (fun local -> lock (monitor) (fun () -> sum := !sum + local))) |>
             ignore
    !sum * steps
```

Our benchmark shows that this version is 7-8× faster than the sequential counterpart. Here we use a `Partitioner` to divide the work into many fixed-size chunks. The size of these chunks should be considerably large to be worth spawning new tasks; as the number of locks is equal to the number of chunks, this number should be small enough. The parallel construct here is **Parallel.ForEach**, the companion of **Parallel.For**, which is employed for non-indices collections. As can be seen from the example, **Parallel.ForEach** and `Partitioner` are really helpful because they provide a very fine-grained control over how parallelization is done. However, **Parallel.For(Each)** is designed for .NET itself. In order to use them conveniently in a functional programming language, one should wrap them in functional constructs and provide parameters for tweaking parallelism.

Comparison of execution time is presented in Table 2.1, and detailed source code can be found in Appendix A.1.

Besides those primitives which have been illustrated in the above examples, TPL also has some mechanisms to provide fine-grained parallel execution:

	LINQ	PLINQ	Sequential	Parallel.For	Parallel.ForEach
Speedup	1x	3-4x	2x	6-8x	14-16x

Table 2.1: Speedup factors of π calculation compared to the slowest version.

- **Degree of Parallelism**
  In **PLINQ**, controlling degree of parallelism is specified by `WithDegreeOf Parallelism(Of TSource)` operator, this operator determines the maximum number of processors used for the query. In **Parallel.For(Each)**, we can set the maximum number of threads for executing a parallel loop by `ParallelOptions` and `MaxDegreeOfParallelism` constructs. These primitives are introduced to avoid spawning too many threads for computation, which leads to too many overheads and slows down computation.

- **Ending Parallel.For(Each) loop early**
  With **Parallel.For(Each)**, if iterating through the whole loop is not needed, we can use `Stop()` and `Break()` functions of `ParallelLoopState` to jump out of the loop earlier. While `Stop()` terminates all indices which are running, `Break()` only terminates other lower indices so the loop will be stopped more slowly than the former case. This early loop-breaking mechanism is sometimes helpful to design parallel-exist or parallel-find functions which are difficult to do with **PLINQ**.

To conclude, **PLINQ** and **Parallel.For(Each)** provide convenient ways to perform data parallelism in .NET platform. While **PLINQ** is declarative and natural to use, we only have little control to how parallelization is done there. With **Parallel.For(Each)**, we have more control over partitioning and distributing workloads on processing units, however their paradigm is somehow imperative. One thing to keep in mind is that parallelism constructs are expensive, so data parallelism is worth performing only when workloads are large and the number of items is big enough.

2.2.3 Task parallelism

While data parallelism focuses on data and the way to distribute them over processing units, task parallelism focuses on how to split tasks into small subtasks which can be executed in parallel. With this point of view, task parallelism is very much related to divide-and-conquer techniques where a big problem is divided into subproblems, and solving subproblems leads to a solution for the original problem. We demonstrate this by the MergeSort algorithm in order to show that task parallelism is relevant and helpful to derive parallel variants

using divide-and-conquer techniques. A simple variant of MergeSort is written below:

```
let rec msort ls =
        if Array.length ls <= 1 then ls
        else
                let ls1, ls2 = split ls
                let ls1', ls2' = msort ls1, msort ls2
        merge (ls1', ls2')
```

The algorithm consists of three steps: the first step involves in splitting the array into two equal-size chunks, the second one is running MergeSort on two halves of the array (which could be done in parallel) and the last step is merging two sorted halves. One interesting thing about parallelizing this algorithm is the important role of **merge** function. If the **merge** function is implemented by comparing first elements of two arrays and gradually putting the smaller element to a new array, the cost of the whole algorithm is dominated by the cost of the **merge** function which is inherently sequential. Some attempts to parallelize MergeSort in this context did not lead to any good results. If the **merge** function is designed using divide-and-conquer techniques, the possibility of parallelism is much bigger as it can be seen in the following code fragment:

```
let rec merge (l1: _ [], l2) =
    if l1 = [||] then l2
    elif l2 = [||] then l1
    elif Array.length l1 < Array.length l2 then merge (l2, l1)
    elif Array.length l1 = 1 then
        if l1.[0] <= l2.[0] then Array.append l1 l2
        else Array.append l2 l1
    else
        let h1, t1 = split l1
        let v1 = h1.[Array.length h1 − 1]
        let idx = binarySearch (l2, v1, 0, l2.Length)
        let m1, m2 = merge (h1, l2.[..idx−1]), merge (t1, l2.[idx..])
        Array.append m1 m2
```

Divide-and-conquer techniques are employed here as follows:

- Divide the longer array into two halves, and these two halves separated by the value of x which is the last element in the first half.
- Split the other array by the value of x using a binary search algorithm. Two first halves of two arrays whose elements are smaller than or equal to x could be safely merged, similarly with two second halves whose elements bigger than x.

- These two merged arrays are concatenated to get the final result.

In parallelism's point of view, two calls of the **merge** function here are independent and easy to parallelize. One rough version of parallel MergeSort is sketched as follows:

```
let rec pmsort ls =
    if Array.length ls <= 1 then ls
    else
        let ls1, ls2 = split ls
        let ls1' = Task.Factory.StartNew(fun () -> pmsort ls1)
        let ls2' = pmsort ls2
        pmerge (ls1'.Result, ls2')
```

where **pmerge** is a parallel variant of **merge** with similar way of parallelization. The idiom used here is the *Future* pattern, and **ls1'** is a future value which will be executed if its `Result` property is invoked later on. In our example, **ls1'.Result** is always executed; therefore, two tasks are always created and executed in parallel. To explicitly wait for all tasks to complete, we use `Task.WaitAll` from `System.Threading.Tasks` module. Here `Task.Factory.StartNew` creates a task for running computation by wrapping around a function closure. For example, we have a similar code fragment as the *Future* pattern:

```
let ls1' = Task.Factory.StartNew(fun () -> pmsort ls1)
let ls2' = Task.Factory.StartNew(fun () -> pmsort ls2)
Task.WaitAll(ls1', ls2')
pmerge (ls1'.Result, ls2'.Result)
```

Our rough parallel version shows that the CPU usage is better but overall execution time is not improved compared to the sequential variant. The reason is that tasks creation is always done regardless of the size of input arrays, which leads to spawning too many tasks for just doing trivial jobs. Thus, we introduce one way to control degree of parallelism:

```
let rec pmsortUtil (ls, depth) =
    if Array.length ls <= 1 then ls
    elif depth = 0 then msort ls
    else
        let ls1, ls2 = split ls
        let ls1' = Task.Factory.StartNew(fun () -> pmsortUtil (ls1, depth-1))
        let ls2' = pmsortUtil (ls2, depth-1)
        pmergeUtil (ls1'.Result, ls2', depth)

let DEPTH = 4
let rec pmsort1 ls =
    pmsortUtil(ls, DEPTH)
```

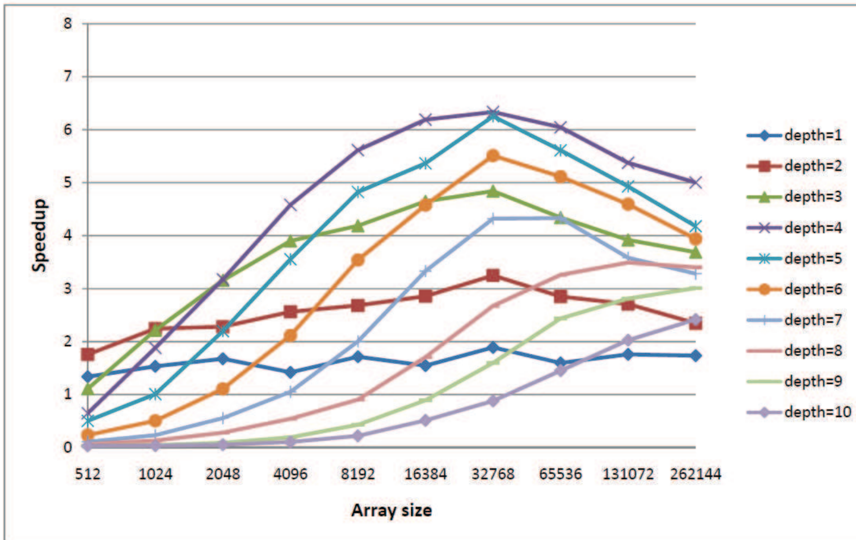


Figure 2.3: Speedup factors of MergeSort algorithm.

Parallelism is only done until a certain depth. When the size of the array is rather small, we fall back to the sequential version. Benchmarking done with different combinations of depth and array size gives us results described in Figure 2.3. Detailed source code can be found in Appendix A.2.

In the Experiment with randomly generated arrays, speedup factors recorded for the size of the array increased from 2^9 to 2^{18} by a factor of 2 and for the depth increased from 1 to 10. The results show that the overall best speedup is recorded for depth equal to 4. And with this configuration, speedup factors are 3-6.5x if array size is greater than or equal 2^{11}. The figure also illustrates that speedup factors could be smaller than 1, it happens when the array size is considerably small compared to overheads of spawned tasks.

Some other points related to `Tasks` are also worth mentioning:

- **Waiting for any task to complete**
  In case of speculative computation or searching, only the first completed task is needed. Therefore, `Task.WaitAny` is used to specify the return.

- **Cancelling running tasks**
  Users can control task cancellation easily with `CancellationTokenSource` class. When a task is cancelled, it throws an `AggregateException` to

indicate its successful termination.

- **Task Graph Pattern**
  When many tasks are created with sophisticated relations between each other, PFX provides a means of specifying graphs of tasks and optimizing them for parallel execution. For example, a task which needs to wait for results from another task is indicated by `Task.ContinueWith` construct. If one task depends on a number of tasks, `Task.Factory.ContinueWhenAll` is utilized.

To sum up, `System.Threading.Tasks` module consists of many helpful constructs for task parallelism. Task parallelism inherently suits to divide-and-conquer algorithms. Using `Task` is rather simple; however, to achieve good efficiency, tasks should be created only when computation is heavier than overheads of creating and coordinating tasks.

2.2.4 Asynchronous computation as parallelization

F# has very good support of asynchronous computation by the notion of *Asynchronous Workflow*. This component is built for F# only and has convenient use by the syntax of `async{...}`. Certainly, *Asynchronous Workflow* best fits IO-bound computation where tasks are executed and are not blocked to wait for the final result. Using asynchronous computation, operations are going to run asynchronously, and resources will not be wasted. In some cases, asynchronous computation could bring benefits to CPU-bound tasks as the following example of computing first 46 Fibonacci numbers:

```
let rec fib x = if x < 2 then 1 else fib (x - 1) + fib (x - 2)

let sfibs = Array.map (fun i -> fib(i)) [|0..45|];;

let pfibs =
    Async.Parallel [| for i in 0..45 -> async { return fib(i) } |]
    |> Async.RunSynchronously;;
```

In this example, functions are turned to be asynchronous by using `async` keyword. While these functions are marked for parallel execution by `Async.Parallel` and `Async.RunSynchronously`, .NET framework will do the job of distributing them to available cores and running them effectively. Our benchmark shows that the asynchronous variant is 2.5× faster than the sequential one. A better speedup can be achieved by using `Task` instead of *Asynchronous Workflow*, however, it is a good idea to use *Asynchronous Workflow* if the computation

consists of both CPU-bound tasks and IO-bound tasks where using `Task` only could be bad due to the blocking property of IO operations. With the presence of *Asynchronous Workflow*, we show that F# has very rich support of parallel constructs. There are many primitives to employ for each algorithm, choosing constructs really depends on the nature of problem and operations required.

2.2.5 Other parallel constructs

The Array.Parallel and PSeq Module

The `Array.Parallel` module is a thin wrapper around `Parallel.For(Each)` constructs, the purpose is providing a convenient interface for parallel operations on `Array`. `Array.Parallel` is easy to use because its signature is exactly the same as `Array` correspondent. Employing this module, we can clearly see the benefit of wrapping imperative parallel constructs by high order functions to hide side effects.

Similar to above module, the `PSeq` module based on PLINQ constructs to provide high order functions for parallel processing on sequences. Input for this module's functions could be sequences such as `Array`, `List` and `Sequence`. At first they are converted to `ParallelEnumerable` and all operations are done on this internal representation. If results in representation of sequences are needed, we need to convert from `ParallelEnumerable`. Therefore, if operations are in a form of aggregation, it is ideal to use `PSeq` as we can avoid the cost of converting to normal sequences. While `Array.Parallel` is available from F# core libraries, `PSeq` is developed as a part of F# PowerPack, an advanced extension from F# team for many mathematics and parallel operations.

Concurrent Data Structures

`Task` is the primitive to work with parallel processing in .NET platform. Internally, the PFX is spawning threads to work with `Task`, so using shared data structures between threads is unsafe. One way to avoid data races is acquiring locks to access shared data structures. This approach is expensive and hard to guarantee both correctness and performance. To resolve this, the PFX provides several collections in `System.Collections.Concurrent` which are safe to be used between threads. These collections have locking mechanisms inside to ensure data consistency, also they ensure good performance by using lock-free algorithms whenever possible. Collections available from the PFX consist of `ConcurrentQueue`, a FIFO queue which can be safely used between

threads; `ConcurrentDictionary` which behaves like a normal dictionary and `ConcurrentBag` which is similar to `Set` but allows adding duplicated items.

2.3 Summary

In this chapter, we have presented important concepts of multicore parallelism. We have also learned multicore parallelism by working on a series of familiar examples in F#. The results show that F# is suitable for fast prototyping parallel algorithms; however, further tuning is essential to achieve high scalability.

Chapter 3

Functional paradigm and multicore parallelism

In this chapter, we study parallel algorithms in the context of a functional programming language. The purpose is finding a cost model of parallel functional algorithms which well represents them in the notion of time complexity and parallelism factor. Other than that, we investigate some hindrances one has to overcome while working with functional parallelism. The key idea is understanding trade-offs between complexity and scalability and finding solutions for the scalability problem that may happen.

3.1 Parallel functional algorithms

In this section, we investigate parallel functional algorithms and understand their efficiency, expressiveness and pitfalls. There are many attempts to research parallel functional algorithms, and NESL, an implicit parallel functional language was proposed for studying those algorithms [3]. We do not focus on presenting detailed features of the language but we employ its computation model for analyzing parallel functional algorithms.

NESL has shown some interesting points about functional parallelism:

- The language relies on a small set of features which are necessary for parallelism, so the core language is more compact and easier to learn.
- NESL has been used to teach parallel programming for undergraduates and the language has been proved to be expressive and elegant due to its functional style.
- The language has been employed to express various parallel algorithms, and recorded speedup factors and execution time have been shown to be competitive with other languages.

Following NESL, many other functional programming languages have been extended to support parallelism. Concurrent ML, an extension of SML/NJ, focuses on supporting various communication primitives. It demonstrates noticeable speedups and good scalability in some examples [12]. Other than that, the Haskell community has progressed quite a lot in research of parallelism and concurrency. The Glasgow Haskell Compiler itself has been parallelized, and some Haskell examples show good speedups on a small number of cores[20].

NESL has demonstrated that the functional paradigm is elegant and helps developers easy to catch up with parallel programming, but one of the biggest contribution of the language is stressing the requirement of a language-based cost model in the context of parallel programming [3]. Traditionally, PRAM (Parallel Random Access Machine) is used as the cost model of parallel programming. PRAM extends the RAM model with the assumption that all processors can access memory locations in a constant time. When we come to the multicore era, the assumption is obviously not true because the cost of accessing local memory locations and remote memory locations are so different. PRAM model is viewed as a virtual machine which is abstracted from real ones, and the model is easy to program but results are somehow difficult to predict for a particular machine. NESL's author, Belloch emphasizes the requirement of using a language-based cost model which is independent from machines but reliable to predict results [3]. The cost model was named the *DAG model of multithreading* due to its illustration by a graph describing computations happening at computation elements. The DAG model consists of two basic elements: *Work*, the total amount of computations which have to be done and *Span*, the longest path of sequential computations in the algorithm. The expressiveness of NESL helps to reason about performance of parallel programs on the DAG model, and the model is simple and intuitive but reliable in performance analysis of parallel programs. Similar to other parallel cost models, it does not account communication costs of parallel programming. Certainly, it is difficult to analyze communication costs but the DAG model is precise enough to predict maximal parallelism of algorithms.

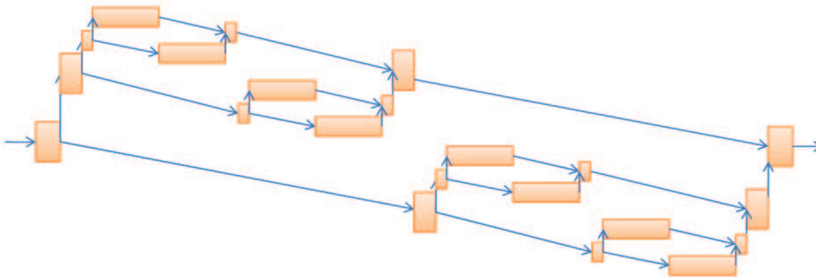


Figure 3.1: Total computations of MergeSort algorithm [29].

Now we shall study performance analysis of some parallel algorithms by adopting this cost model. In the graph representing an algorithm, *Work* is the sum of all computations and *Span* is the critical path length or the unavoidable sequential bottleneck in the algorithm. Take MergeSort algorithm as an example, *Work* and *Span* are illustrated in Figure 3.1 and 3.2.

Let $\mathtt{W}$, $\mathtt{S}$ and $\mathtt{T_P}$ denote *Work*, *Span* and running time of the algorithm with P processors. The asymptotic running time of the sequential algorithm is $\mathtt{W} = \mathtt{O}(\mathtt{n}\ \mathtt{log}\ \mathtt{n})$. In the **merge** phase, the algorithm uses a sequential merge process that means each element of each half is gradually put into an accumulated buffer, so the operation is linear to the size of the biggest half. It is easy to infer that $\mathtt{S} = \mathtt{O}(\mathtt{n})$ where $\mathtt{n}$ is the size of the input array.

According to Amdahl's Law, the below condition is established:

$$\frac{\mathtt{W}}{\mathtt{T_P}} \leq \mathtt{P}$$

Because total running time is bounded by sequential computation, we have $\mathtt{T_P} \geq \mathtt{S}$ which leads to:

$$\frac{\mathtt{W}}{\mathtt{T_P}} \leq \frac{\mathtt{W}}{\mathtt{S}}$$

The condition shows that speedup factor is limited by the ratio of *Work* and *Span*, namely *parallelism factor*. For a given *Span*, the parallel algorithm should do as much *Work* as possible. On the contrary, when *Work* is unchanged we should shorten the critical path length in the graph of computation. In general, the ratio $\mathtt{W}\ /\ \mathtt{S}$ provides an easy and effective way to predict performance of parallel algorithms. In the case of above MergeSort algorithm, the maximal parallelism is $\mathtt{O}(\mathtt{log}\ \mathtt{n})$. This parallel algorithm is inefficient because roughly speaking, given `1024` elements of computation we only get maximum

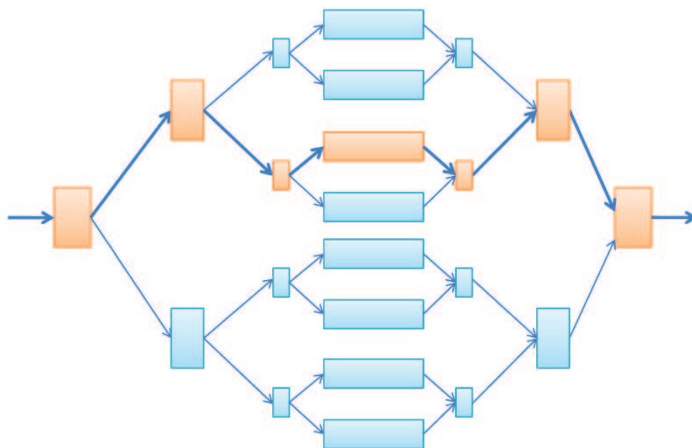


Figure 3.2: Critical path length of MergeSort algorithm [29].

10× speedup regardless of the number of used processors. Usually, analysis of parallel algorithm is a good indication of its performance in practice. Certainly, parallelism factor of MergeSort algorithm should be improved before implementation. A consideration could be replacement of sequential merging by parallel merging as it has been described in Section 2.2.3. The divide-and-conquer nature of parallel merging helps to reduce *Span* to $\mathtt{O}(\mathtt{log}^2\ \mathtt{n})$, so the parallelism factor is $\mathtt{O}(\mathtt{n}\ /\mathtt{log}\ \mathtt{n})$ which is much better than that of the original algorithm. This parallelism factor also confirms our experience of using parallel merging effectively (see Figure 2.3).

Take a look at our example of calculating π, time complexity of the sequential algorithm is $\mathtt{O}(\mathtt{n})$, the **Select** phase results in the critical path length of $O(1)$ and the **Sum** phase has the critical path length of $\mathtt{O}(\mathtt{log}\ \mathtt{n})$. Therefore we have a *Span* of $\mathtt{O}(\mathtt{log}\ \mathtt{n})$ and a *parallelism factor* of $\mathtt{O}(\mathtt{n}\ /\mathtt{log}\ \mathtt{n})$. We can see that the parallel algorithm is efficient; therefore, a good speedup is feasible in practice.

3.2 Some pitfalls of functional parallelism on the multicore architecture

As discussed in Chapter 2, F# and functional programming languages in general are suitable for fast prototyping parallel algorithms. The default immutability helps developers to ensure the correctness of implementation at first, and par-

allel implementation is obtained with a little effort when making use of elegant parallel constructs. However, the efficiency of these implementations is still an open question. This section discusses some issues of functional parallelism on the multicore architecture and possible workarounds in the context of F#.

Memory Allocation and Garbage Collection

High-level programming languages often make use of a garbage collector to discard unused objects when necessary. In these languages, users have little control over memory allocation and garbage collection; in the middle of execution of some threads if the garbage collector needs to do its work, it is going to badly affect performance. The advice for avoiding garbage collection is allocating less which leads to garbage collecting less by the runtime system. However, for some memory-bound applications it is difficult to reduce the amount of allocated memory. The problem of memory allocation is more severe in the context of functional programming. Function programming promotes usage of short-lived objects to ensure immutability, and those short-lived objects are created and destroyed frequently, which leads to more work for garbage collectors. Also some garbage collectors are inherently sequential and preventing threads to run in parallel, Matthew *et al.* stated that sequential garbage collection is a bottleneck to parallel programming in Poly/ML [20]. F# inherits the garbage collector (GC) from .NET runtime, which runs in a separate thread along with applications. Whenever GC needs to collect unused data, it suspends all others' threads and resumes them when its job is done. .NET GC is running quite efficiently in that concurrent manner; however, if garbage collection occurs often, using the Server GC might be a good choice. The Server GC creates a GC thread for each core so scalability of garbage collection is better [19]. Come back to two examples which have been introduced in Chapter 2, one of the reasons for a linear speedup of π calculation (see Table 2.1) is no further significant memory allocation except the input array. However, a sublinear speedup of MergeSort algorithm (see Figure 2.3) could be explained by many rounds of garbage collection occurring when immediate arrays are discarded.

False Cache-line Sharing

When a CPU loads a memory location into cache, it also loads nearby memory locations into the same cache line. The reason is to make the access to this memory cell and nearby cells faster. In the context of multithreading, different threads writing to the same cache line may result in invalidation of all CPUs' caches and significantly damage performance. In the functional-programming

setting, false cache-line sharing is less critical because each value is often written only once when it is initialized. But the fact that consecutive memory allocations make independent data fall into the same cache line also causes problem. Some workarounds are padding data which are concurrently accessed or allocating memory locally in threads.

We illustrate the problem by a small experiment as follows: an array which has a size equal to the number of cores and each array element is updated 10000000 times [25]. Because the size of the array is small, its all elements tend to fall into the same cache line and many concurrent updates to the same array will invalidate the cache line many times and badly influence the performance. The below code fragment shows concurrent updates on the same cache line:

```
let par1() =
    let cores = System.Environment.ProcessorCount
    let counts = Array.zeroCreate cores
    Parallel.For(0, cores, fun i ->
                    for j = 1 to 10000000 do
                        counts.[i] <- counts.[i] + 1)|> ignore
```

The measurement of sequential and parallel versions on the 8-core machine is shown as follows:

```
> Real: 00:00:00.647, CPU: 00:00:00.670, GC gen0: 0, gen1: 0, gen2: 0 // sequential
> Real: 00:00:00.769, CPU: 00:00:11.310, GC gen0: 0, gen1: 0, gen2: 0 // parallel
```

The parallel variant is even slower than the sequential one. We can fix the problem by padding the array by garbage data, this approach is 17× faster than the naive sequential one:

```
let par1Fix1() =
    let cores = System.Environment.ProcessorCount
    let padding = 128 / sizeof<int>
    let counts = Array.zeroCreate ((1+cores)*padding)
    Parallel.For(0, cores, fun i ->
                    let paddedI = (1 + i) * padding
                    for j = 1 to 10000000 do
                        counts.[paddedI] <- counts.[paddedI] + 1
                        )|> ignore
```

```
> Real: 00:00:00.038, CPU: 00:00:00.530, GC gen0: 0, gen1: 0, gen2: 0
```

Another fix could be allocating data locally in the thread, so false cache-line sharing cannot happen. This version is also 14× faster than the sequential counterpart:

```
let par1Fix2() =
    let cores = System.Environment.ProcessorCount
    let counts = Array.zeroCreate cores
    Parallel.For(0, cores, fun i ->
                    counts.[i] <- Array.zeroCreate 1
                    for j = 1 to 10000000 do
                        counts.[i].[0] <- counts.[i].[0] + 1)|> ignore

> Real: 00:00:00.044, CPU: 00:00:00.608, GC gen0: 0, gen1: 0, gen2: 0
```

Locality Issues

Changing a program to run in parallel can affect locality of references. For example, when data allocated in the global scope are distributed to different threads on different cores, some threads have to access remote caches to be able to fetch data which negatively influences the performance. Also when the garbage collector finishes its job, memory layout may have been changed and cache locality is destroyed. It is usually difficult to ensure good cache locality in a high-level language because memory allocation and deallocation is implicit to users and compilers are responsible for representing data in memory in an optimal way. In the functional-programming setting, the situation will be remedied a little by using mutation (e.g. by employing *Array-based* representation) but the solution is complex and unscalable if one has to deal with recursive data structures.

Some algorithms which have bad memory access patterns demonstrate poor cache locality regardless of their data representation. There is a lot of research on cache complexity of algorithms on the multicore architecture, especially a class of algorithms which is important in the multicore era, namely *cache-oblivious algorithms*. The cache-oblivious model was proposed by Prokop [26]. Before Prokop's work, algorithms and data structures were designed in a cache-aware (cache-conscious) way to reduce the ratio of cache misses, for example, B-tree is a well-known example of cache-aware data structures in which the parameter B is tuned by using the CPU cache size. Cache-oblivious algorithms recursively divide a problem into smaller parts and do computation on each part. Eventually subproblems fit into cache and significant amount of computations are done without causing any cache miss. We demonstrate cache-oblivious algorithms by an example of matrix multiplication; the ordinary algorithm follows the following formula:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

which can be represented by the pseudocode:

```
Naive-Mult(A, B, C, n)
  for i = 1 to n do
    for j = 1 to n do
          do c_ij = 0
          for k = 1 to n do
            c_ij = c_ij + a_ik b_kj
```

The *Naive-Mult* algorithm incurs $\Omega(n^3)$ cache misses if matrices are stored in the row-major order. Strassen's matrix multiplication algorithm works in a recursive manner on their four submatrices:

$$
\begin{aligned}
P_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22}) \times B_{11} \\
P_3 &= A_{11} \times (B_{12} - B_{22}) \\
P_4 &= A_{22} \times (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12}) \times B_{22} \\
P_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 - P_2 + P_3 + P_6
\end{aligned}
$$

Prokop proved that Strassen's algorithm is cache-oblivious and it incurs an amount of cache misses which is an order of $\Omega(\sqrt{Z})$ less than that of the naive algorithm (where Z is the size of matrix which fits in cache).

We have benchmarked sequential and parallel variants of the two algorithms on the same machine with different sizes of matrices. Speedup factors of parallel versions compared to sequential ones and those between Strassen's sequential version and a naive sequential version are recorded in Figure 3.3.

There are some conclusions deduced from the benchmark results. First, Strassen's algorithms exhibit close-to-linear speedups which are better than naive ones. Given the fact that matrix multiplication is an embarrassing-parallel problem, non-optimal speedup of naive algorithms may blame the significant amount of incurred cache misses which are really expensive in the context of multicore parallelism. Moreover, the sequential version of Strassen also surpasses the naive sequential one in terms of performance, and the magnitude of speedup increases when the size of matrices increases. The results illustrate that cache-oblivious

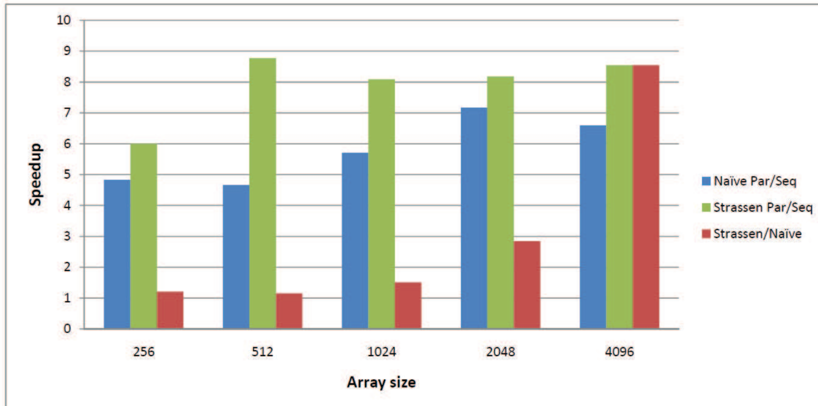


Figure 3.3: Speedup factors of matrix multiplication algorithms.

algorithms are not only good in parallel execution, but also their sequential variants perform well on the multicore architecture.

Cache-oblivious algorithms are independent of CPU cache sizes, working well on any memory hierarchy and proved to be optimal in cache complexity. On the rise of multicore parallelism, cache-oblivious algorithms may play an important role in deriving efficient parallel programs. The superior performance of cache-oblivious algorithms raises the need of proposing such algorithms for the multicore architecture. However, not so many cache-oblivious algorithms are found, and some of cache-oblivious variants are too complex which makes them less efficient to be used in practice. Blelloch *et al.* also shows that divide-and-conquer nature and cache-oblivious model are working well on flat data structures (array, matrix), but results on recursive data structures are still to be established [4].

3.3 Summary

In this chapter, we have chosen the *DAG model of multithreading* as the cost model of multicore parallelism. This model is going to be used for analyzing parallel algorithms from now on. The pitfalls we have presented are common for parallel algorithms in functional programming languages, and understanding them helps us explain behaviours of parallel implementations and find workarounds which can remedy the situation. In next chapters, we focus on describing the problem of our interest and discussing its parallel algorithms.

Chapter 4

Theory of Presburger Arithmetic

In this chapter, we discuss Presburger Arithmetic and its properties. A lot of research has been conducted to decide Presburger fragments. We present two decision procedures including Cooper's algorithm and the Omega Test, and they play important roles in processing Presburger fragments of our interest later on.

4.1 Overview

Presburger Arithmetic introduced by Mojzaesz Presburger in 1929 is a first-order theory of integers which accepts + as its only operation. An expression is considered to be a Presburger formula if it contains elements in the following forms:

- $s = t, s \neq t, s < t, s > t, s \leq t, s \geq t$ Comparison constraints
- $d \mid t$ Divisibility constraints
- $\top$ (true), $\bot$ (false) Propositional constants
- $F \lor G, F \land G, \neg F$ Propositional connectives
- $\exists\, x.\, F, \forall\, x.\, F$ First-order fragments

where s and t are terms, d is an integer and x is a variable. Terms consist of integer constants and variables, they accept addition (+), subtraction (-) and multiplication by constants. One adopted convention is abbreviation of $\mathtt{Q}\ \mathtt{x}_1.\mathtt{Q}\ \mathtt{x}_2...\mathtt{Q}\ \mathtt{x}_n.\ \mathtt{F}$ as $\mathtt{Q}\ \mathtt{x}_1\mathtt{x}_2...\mathtt{x}_n.\ \mathtt{F}$.

For example, a classic example of representing some amount of money by 3-cent coins and 5-cent coins appears in PA as follows:

$$\forall z.\exists x\ \exists y.\ 3x + 5y = z$$

$$\forall z.\ z \geq 8 \Rightarrow \exists x\ \exists y.\ 3x + 5y = z$$

Or the clause shows existence of even numbers could be formulated:

$$\exists x.\ 2 \mid x$$

Presburger had proved Presburger Arithmetic to be *consistent*, *complete* and *decidable* [31]. The *consistency* means there is no Presburger formula so that both that formula and its negation can be deduced. The *completeness* shows that it is possible to deduce any true Presburger formula. The *decidable* property states that there exists an algorithm which decides whether a given Presburger statement is true or false; that algorithm is called a decision procedure for PA. An extension of Presburger with multiplication of variables is however undecidable, Presburger showed a counterexample of deciding the statement $\exists\ x\ y\ z.\ x^2 + y^2 = z^2$, which is a special case of Fermat's last theorem [31].

After Presburger's work, PA attracts a lot of attention due to its application in different areas. In 1972, Cooper proposed a complete algorithm for deciding PA formulas which named after him, he found application of PA in automatic theorem proving [6]. Some works by Bledsoe and Shostak introduced a new algorithm for Presburger fragments resulted from program validation [2, 30]. In 1992, the Omega Test, another complete algorithm for PA, was invented

by Pugh who used his algorithm in dependence analysis in the context of production compilers [27]. There is an automata-based approach which represents Presburger formulas by means of automata and transforms these automata for quantifier elimination [8], but we do not consider this approach in our report. Another work recognized the appearance of PA in the model checking problem which also raised the requirement of efficient decision procedures for PA [10]. However, the complexity of those decision procedures appears to be very challenging. Let n be the length of the input PA formula; the worst-case running time of any decision procedure is at least $2^{2^{cn}}$ for some constant $c > 0$ [9]. Also Oppen proved a triply exponential bound $2^{2^{2^{cn}}}$ for a decision procedure for PA, namely Cooper's procedure [24]. PA is an interesting case where its decision procedures require more than exponential time complexity, how to handle these formulas efficiently is still an open question.

4.2 Decision Procedures for Presburger Arithmetic

In this section, we introduce two decision algorithms for PA: Cooper's procedure and the Omega Test. These procedures are chosen due to both their effectiveness in solving PA formulas and their completeness in dealing with the full class of PA. Such decision algorithms for a small subset of PA exist, for example, Bledsoe proposed the SUP-INF algorithm for proving PA formulas with only universal quantifiers in Prenex Normal Form (PNF) [2]. The algorithm then was improved by Shostak to support both deciding validity and invalidity of this subset of formulas [30]. The SUP-INF method is believed to have 2^n worst-case time complexity and helpful for PA fragments generated from program validation [30].

4.2.1 Cooper's algorithm

The algorithm was invented by Cooper in 1972 [6], a detailed discussion of the algorithm is presented below. Given a total ordering $\prec$ on variables, every term can be normalized to the following equivalent form:

$$\mathtt{c_1} \cdot \mathtt{x_1} + \ldots + \mathtt{c_n} \cdot \mathtt{x_n} + \mathtt{k}$$

where $\mathtt{c_i} \neq 0$ and $\mathtt{x_1} \prec \mathtt{x_2} \prec \ldots \prec \mathtt{x_n}$. The above form is called a *canonical* form in which a term has a unique representation. Consider a formula $\exists x.F(x)$ where $F(x)$ is quantifier free. Cooper's procedure for quantifier elimination can be described as follows [5]:

Step 1. Preprocess the formula

Put the formula into negation normal form (NNF) where negation only occurs in literals by using De Morgan's laws:

$$\neg(\mathtt{F} \wedge \mathtt{G}) \equiv \neg\mathtt{F} \vee \neg\mathtt{G}$$

$$\neg(\mathtt{F} \vee \mathtt{G}) \equiv \neg\mathtt{F} \wedge \neg\mathtt{G}$$

All literals are ensured in the forms:

$$\mathtt{0} = \mathtt{t},\ \mathtt{0} \neq \mathtt{t},\ \mathtt{0} < \mathtt{t},\ \mathtt{i} \mid \mathtt{t},\ \neg(\mathtt{i} \mid \mathtt{t})$$

Notice that negation only happens in divisibility constraints. Transformation of comparison constraints can be done by applying following rules:

$$\begin{array}{lcl} \mathtt{0} \leq \mathtt{t} & \equiv & \mathtt{0} < \mathtt{t} - \mathtt{1} \\ \mathtt{0} \geq \mathtt{t} & \equiv & \mathtt{0} < -\mathtt{t} - \mathtt{1} \\ \mathtt{0} > \mathtt{t} & \equiv & \mathtt{0} < -\mathtt{t} \\ \neg(\mathtt{0} < \mathtt{t}) & \equiv & \mathtt{0} < -\mathtt{t} - \mathtt{1} \end{array}$$

Step 2. Normalize the coefficients

Let δ denote the least common multiple (lcm) of all coefficients of x, normalize all constraints so that coefficients of x are equal to δ. The resulting formula is $G(\delta x)$, and we have the normalized formula in which every coefficient of x is 1:

$$\mathtt{F}'(\mathtt{x}') = \mathtt{G}(\mathtt{x}') \ \wedge\ \delta \mid \mathtt{x}'$$

Step 3. Construct an equivalent quantifier-free formula

There are two quantifier-free formulas which correspond to $F'(x)$, either in left infinite projection or in right infinite projection:

$$\mathtt{F}''(\mathtt{x}) = \bigvee_{\mathtt{j}=\mathtt{1}}^{\delta} (\mathtt{F}'_{-\infty}(\mathtt{j}) \vee \bigvee_{\mathtt{b} \in \mathtt{B}} \mathtt{F}'(\mathtt{b} + \mathtt{j}))$$

$$\mathtt{F}''(\mathtt{x}) = \bigvee_{\mathtt{j}=\mathtt{1}}^{\delta} (\mathtt{F}'_{+\infty}(\mathtt{j}) \vee \bigvee_{\mathtt{a} \in \mathtt{A}} \mathtt{F}'(\mathtt{a} - \mathtt{j}))$$

α	$\alpha_{-\infty}$	$\alpha_{+\infty}$
$0 = x + t'$	$\bot$	$\bot$
$0 \neq x + t'$	$\top$	$\top$
$0 < x + t'$	$\bot$	$\top$
$0 < -x + t'$	$\top$	$\bot$
α	α	α

Table 4.1: Construction of infinite formulas.

Construction of infinite formulas $\mathtt{F}'_{-\infty}(\mathtt{j})$ and $\mathtt{F}'_{+\infty}(\mathtt{j})$ is based on substitution of every literal α by $\alpha_{-\infty}$ and $\alpha_{+\infty}$ respectively:

The inner big disjunct of formulas is constructed by accumulating all literals which are A-Terms or B-Terms:

Literals	B	A
$0 = x + t'$	$-t' - 1$	$-t' + 1$
$0 \neq x + t'$	$-t'$	$-t'$
$0 < x + t'$	$-t'$	$-$
$0 < -x + t'$	$-$	t'
α	$-$	$-$

Table 4.2: Construction of A-Terms and B-Terms [6].

There are some bottlenecks of Cooper's algorithm which need to be carefully addressed. First, coefficients of big disjuncts become quite huge after some quantifier alternation. One way to avoid this problem is keeping big disjuncts symbolically and only expanding them when necessary. Actually, symbolic representation of big disjuncts is easy to implement and quite efficient because inner formulas demonstrate all properties of big disjuncts. Another optimization proposed by Reddy *et al.* is only introducing a new divisibility constraint after selecting a literal for substitution. This modification avoids calculating lcm of all coefficients of a variable which is often a huge value so that each substitution results in a new coefficient smaller than lcm [28]. The authors argued that if quantifier alternation is done for a series of existential quantifiers only, *heterogeneous coefficients* of different divisibility constraints are not multiplied together; therefore, coefficients are not exploded. Another bottleneck of Cooper's algorithm is a large number of A-Terms and B-Terms occurring while doing quantifier alternation. Due to symmetric projections, we can always choose an infinite projection with the least number of literals. This heuristic minimizes the number of disjuncts inside the bounded formula. However, when quantifiers are eliminated, increase of the number of literals is unavoidable. Another technique

to minimize the number of literals inside quantifier alternation is eliminating blocks of quantifiers [5]. Based on the following rule:

$$\exists x_1 ... x_{n-1}. \bigvee_{j=1}^{\delta} (F'_{-\infty}(x_1 ... x_{n-1}, j) \vee \bigvee_{b \in B} F'(x_1 ... x_{n-1}, b + j))$$

$$\equiv \bigvee_{j=1}^{\delta} (\exists x_1 ... x_{n-1}. F'_{-\infty}(x_1, ..., x_{n-1}, j) \vee \bigvee_{b \in B} \exists x_1 ... x_{n-1}. F'(x_1 ... x_{n-1}, b + j))$$

we can see that after each quantifier alternation, only $1 + |B|$ formulas need to be examined in the next iteration. Therefore, removing a series of quantifiers does not increase sizes of quantifying formulas.

Due to the huge time complexity of PA, many heuristics are incorporated to deal with subsets of PA. For example, for PA fragments with universal quantifiers only, a heuristic is invalidating $\forall \overline{x}. F(\overline{x})$ with a particular *false* instance $F(c)$ [17]; some ground values of the formula have been instantiated to quickly decide the formula. A variant of Cooper's procedure with this simple technique is tested in 10000 randomly generated formulas and shown to outperform some other decision procedures. Another heuristic is solving a set of divisibility constraints [5]. When eliminating quantifiers, formulas could be represented symbolically, but they have to be expanded for evaluation. And this expansion is expensive on even a small problem; the idea of solving divisibility constraints is instead of expanding big disjuncts by all possible assignments, only assignments which satisfy divisibility constraints are instantiated. This heuristic is relevant because each quantifier alternation generates a new divisibility constraint in big disjuncts.

4.2.2 The Omega Test

The Omega Test proposed by Pugh [27] is an extension of Fourier-Motzkin variable elimination to check dependence analysis in a production compiler. It consists of a procedure for eliminating equalities and elements for deciding satisfiability of a conjunction of weak inequalities. Three elements of the Omega Test are the *Real Shadow*, the *Dark Shadow* and the *Gray Shadow* which are summarized in Figure 4.1.

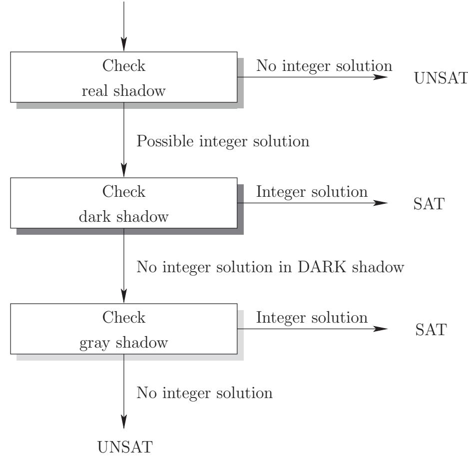


Figure 4.1: Overview of the Omega Test [18].

Real Shadow

The *Real shadow* is an overapproximating projection of the problem, and it is used for checking unsatisfiability [18]:

$$\exists \mathtt{x}.\ \beta \leq \mathtt{bz} \wedge \mathtt{cz} \leq \gamma \implies \mathtt{c}\beta \leq \mathtt{b}\gamma$$

If the *Real shadow* i.e. the equation $\mathtt{c}\beta \leq \mathtt{b}\gamma$ does not hold, there is no real solution, hence there is no integer solution for the problem.

Dark Shadow

The *Dark shadow* is an underapproximating projection; therefore, it is used to check satisfiability of the problem [18]:

$$\mathtt{b}\gamma - \mathtt{c}\beta \geq (\mathtt{c}-\mathtt{1})(\mathtt{b}-\mathtt{1}) \implies \exists \mathtt{x}.\ \beta \leq \mathtt{bz} \wedge \mathtt{cz} \leq \gamma$$

Notice that the *Dark shadow* i.e. the equation $\mathtt{b}\gamma - \mathtt{c}\beta \geq (\mathtt{c}-\mathtt{1})(\mathtt{b}-\mathtt{1})$ guarantees the existence of an integer between b/β and c/γ. If the *Dark shadow* holds, an integer solution for the problem exists. One nice thing is that if $c = 0$ or $b = 0$, the *Real shadow* and the *Dark shadow* are the same. They are then called the *Exact shadow*, and we have the Fourier-Motzkin elimination for integers:

$$\exists \mathtt{x}.\ \beta \leq \mathtt{bz} \wedge \mathtt{cz} \leq \gamma \iff \mathtt{c}\beta \leq \mathtt{b}\gamma \tag{4.1}$$

The *Exact shadow* is an easy and convenient way to eliminate quantifiers in a subset of problems. Pugh argues that the *Exact shadow* happens quite often in the domain of dependence analysis which makes the Omega Test become a very efficient technique for quantifier alternation [27].

Gray Shadow

The *Gray shadow* is tested when the *Real shadow* is true and the *Dark shadow* is false [18]:

$$\mathtt{c}\beta \leq \mathtt{cbz} \leq \mathtt{b}\gamma \ \wedge \ \mathtt{b}\gamma - \mathtt{c}\beta > (\mathtt{c}-1)(\mathtt{b}-1)$$

After simplification, the above condition can be used to reduce quantifier elimination to check a finite number of candidates of $\mathtt{bz}$: $\mathtt{bz} = \beta + \mathtt{i}$ for $0 \leq \mathtt{i} \leq \frac{\mathtt{cb-c-b}}{\mathtt{b}}$

In general, every quantifier can be eliminated from a Presburger formula in PNF:

$$\mathtt{Q_1x_1.Q_2x_2....Q_nx_n.}\bigvee_{\mathtt{i}}\bigwedge_{\mathtt{j}}\mathtt{L_{ij}}$$

where $Q_k \in \{\exists, \forall\}$ and L_{ij} are literals in a form of weak inequalities.

Quantifiers are removed in a bottom-up manner. The formula $\bigvee_{\mathtt{i}}\bigwedge_{\mathtt{j}}\mathtt{L_{ij}}$ is in disjunctive normal form (DNF). If $\mathtt{Q_n} = \exists$ we have the following equivalence:

$$\exists\mathtt{x_n}.\bigvee_{\mathtt{i}}\bigwedge_{\mathtt{j}}\mathtt{L_{ij}} \equiv \bigvee_{\mathtt{i}}\exists\mathtt{x_n}.\bigwedge_{\mathtt{j}}\mathtt{L_{ij}}$$

Now we can use elements of the Omega Test on the conjunction of literals. On the other hand, any universal quantifier is removed by using the dual relation $\forall\mathtt{x}.\phi(\mathtt{x}) \equiv \neg\exists\mathtt{x}.\neg\phi(\mathtt{x})$ and converting $\neg\phi(x)$ into DNF.

The procedure is done recursively until no quantifier is left.

The Omega Test has some advantages and disadvantages compared to Cooper's algorithm:

- Elements of the Omega Test cannot deal with divisibility constraints. To be able to use the Omega Test for an arbitrary formula, one has to eliminate divisibility constraints first, which is quite costly if they occur often in the formula. Divisibility constraint elimination is beyond the scope of this paper.

- The Omega Test requires the inner formula to be in DNF which can cause the formula to grow very fast compared to NNF of Cooper's algorithm. Moreover, once the *Gray shadow* has to be checked, it generates a disjunction of constraints and the resulting formula is not in DNF anymore. Therefore, even in a block of quantifiers, each quantifier elimination requires to translate the input formula into DNF, which is prohibitively expensive. In contrast, Cooper's algorithm handles blocks of quantifiers quite well, and a formula is normalized into NNF once for each block.
- The Omega Test does not have the problem of huge coefficients as Cooper's algorithm because each shadow only involves in pairs of inequalities.

Cooper's algorithm and the Omega Test have good characteristics to make them competitive with each other. Both algorithms have been implemented in HOL theorem-proving system [22]. The results show that while the Omega Test is a little faster, there are problems where Cooper's algorithm outperforms its competitor. It is also easy to construct formulas more favorable for either one of these procedures. Thus, using both the Omega Test and Cooper's algorithm in a complete procedure might be a good idea. One possible solution is executing two algorithms in a parallel manner [22], and another solution is combining elements of the Omega Test and Cooper's procedure in a clever way. For example, the Omega Test is employed for resolving inequalities so coefficients of inequalities do not contribute to lcm and resulting small disjuncts are resolved by Cooper's algorithm. This combination is incorporated into the SMT-solver Z3, and Z3 has capability to solve PA formulas quite efficiently [1].

4.3 Summary

Presburger Arithmetic is a theory of integer with addition, and it has been proved to be consistent, complete and decidable. However, decision procedures for PA have a huge time complexity with doubly-exponential lower bound and triply-exponential upper bound, we might think that parallel decision procedures can remedy the situation. In the next chapter, we present our Presburger fragments with some specific properties, and parallel execution of decision procedures will be discussed later.

CHAPTER 5

Duration Calculus and Presburger fragments

In this chapter, we briefly discuss Duration Calculus and the model-checking problem. The idea is understanding how it leads to the generation of side-condition Presburger formulas. Patterns of these Presburger fragments are examined to derive a good simplification process and efficient decision procedures. Also we present a simplification algorithm which is based on understanding formulas' patterns and well-formed logic rules.

5.1 Duration Calculus and side-condition Presburger formulas

Interval Temporal Logic (ITL) proposed by Moszkowski [14] considers a formula ϕ as a function from time intervals to truth values. Formulas are generated by the following rules:

$$\phi \ ::= \ \theta_1 = \theta_2 \mid \neg\phi \mid \phi \vee \psi \mid \phi ^\frown \psi \mid (\exists x)\phi \mid ...$$

where *Chop* ($^\frown$) is the only modality of ITL, and its satisfiability is expressed by satisfiability of inner formulas in subintervals. Duration Calculus extends ITL

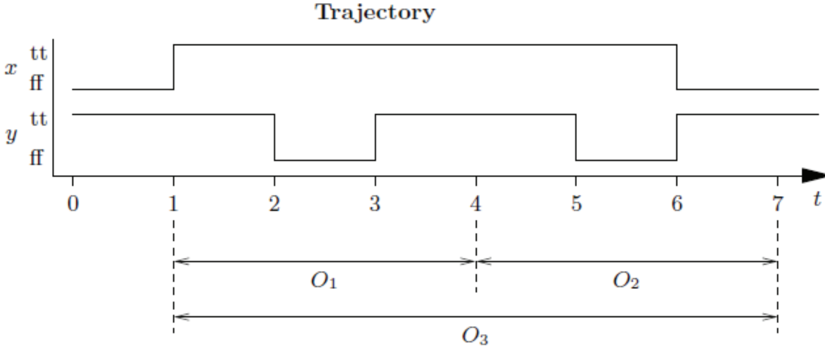


Figure 5.1: Observation intervals [10].

by the notion of durations $\int_b^e S(t)dt$, this extension leads to succinct formation for problems without involving any interval endpoint [14].

For example, in Figure 5.1 we have the observation that $(\int x \geq 3) \wedge (\int y < 3)$ and $\int y - 2\int(x \wedge \neg y) = 0$ hold on [1, 4] and [4, 7] respectively. Therefore, $((\int x \geq 3) \wedge (\int y < 3))^\frown(\int y - 2\int(x \wedge \neg y) = 0)$ holds on [1, 7].

DC is tailored to reason about embedded real-time systems in a highly abstract way. We consider discrete-time DC fragments whose syntax is defined as follows:

$$S ::= 0 \mid 1 \mid P \mid \neg S \mid S_1 \vee S_2$$

$$\phi ::= \top \mid \Sigma_{i\in\Omega} c_i \int S_i \rhd\lhd k \mid \neg\phi \mid \phi \wedge \psi \mid \phi^\frown\psi$$

where P is in the set of state variable names, $k,\ c_i \in Z$ and $\rhd\lhd \in \{<, \leq, =, \geq, >\}$.

The above fragments of DC are undecidable already; if the duration is limited in the form $\int S \rhd\lhd k$, the discrete-time satisfiability problem is decidable [10]. With the decidable fragments of DC, the model-checking problem is described in the following manner: *models are expressed by means of automata (normally, a labelled automata like Kripke structures) and durations are calculated basing on traces of execution.* Here the notion of traces is abstracted to the notion of visiting frequencies of nodes. Therefore, given a DC formula ϕ and a Kripke structure K, K is a model of ϕ when every trace of K satisfies ϕ, written $K \models \phi$. We consider an example of Kripke structures described in Figure 5.2 and check it against the property $\int P < 2$. Certainly the answer is false because there exist some traces which visit P at least twice.

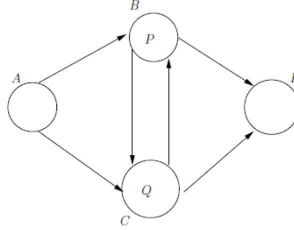


Figure 5.2: A simple Kripke structure [15].

When abstracting from traces, some frequencies are not corresponding to any real trace, so a counting semantic in a multiset m is introduced [10]. The multiset guarantees the consistency between visiting frequencies and traces. The model-checking algorithm does a bottom-up marking and generates side-condition in a form of Presburger formulas. The consistency condition $C(K, i_0, j_0, \overline{m}, \overline{e})$ is expressed by linear constraints of inflow and outflow equations, and there are some additional constraints introduced to deal with loop structures [10]. $C(K, i_0, j_0, \overline{m}, \overline{e})$ has the size of $O(|V| + |E|)$ when every loop has a unique entry point and its satisfiability is the sufficient condition for existence of a consistent trace from i_0 to j_0. The simplified marking algorithm is given in [10] as follows:

$$
\begin{array}{lcl}
sim_t(\top, i, j, \overline{m}, \overline{e}) & = & true \\
sim_f(\top, i, j, \overline{m}, \overline{e}) & = & false \\
\\
sim_t(\Sigma_{i\in\Omega} c_i \int S_i < k, i, j, \overline{m}, \overline{e}) & = & \Sigma_{i\in\Omega} c_i \Sigma_{v\in V, v\models S_i} m[v] < k \\
sim_f(\Sigma_{i\in\Omega} c_i \int S_i < k, i, j, \overline{m}, \overline{e}) & = & \Sigma_{i\in\Omega} c_i \Sigma_{v\in V, v\models S_i} m[v] \geq k \\
\\
sim_t(\neg\phi, i, j, \overline{m}, \overline{e}) & = & sim_f(\phi, i, j, \overline{m}, \overline{e}) \\
sim_f(\neg\phi, i, j, \overline{m}, \overline{e}) & = & sim_t(\phi, i, j, \overline{m}, \overline{e}) \\
\\
sim_t(\phi_1 \wedge \phi_2, i, j, \overline{m}, \overline{e}) & = & sim_t(\phi_1, i, j, \overline{m}, \overline{e}) \wedge sim_t(\phi_2, i, j, \overline{m}, \overline{e}) \\
sim_f(\phi_1 \wedge \phi_2, i, j, \overline{m}, \overline{e}) & = & sim_f(\phi_1, i, j, \overline{m}, \overline{e}) \vee sim_f(\phi_2, i, j, \overline{m}, \overline{e})
\end{array}
$$

and:

$$sim_t(\phi_1 \frown \phi_2, i, j, \overline{m}, \overline{e}) =$$

$$\bigvee_{k \in V} \left(\begin{array}{l} \exists\, \overline{m}_1, \overline{m}_2, \overline{e}_1, \overline{e}_2 : \mu \\ \wedge \\ \forall\, \overline{m}_1, \overline{m}_2, \overline{e}_1, \overline{e}_2 : \mu \Rightarrow sim_t(\phi_1, i, k, \overline{m}_1, \overline{e}_1) \wedge sim_t(\phi_2, k, j, \overline{m}_2, \overline{e}_2) \end{array} \right)$$

$$sim_f(\phi_1 \frown \phi_2, i, j, \overline{m}, \overline{e}) =$$

$$\bigwedge_{k \in V} (\mu \Rightarrow sim_f(\phi_1, i, k, \overline{m}_1, \overline{e}_1) \vee sim_f(\phi_2, k, j, \overline{m}_2, \overline{e}_2))$$

where:

$$\mu = \bigwedge_{v,w \in dom\ m} \left(\begin{array}{c} m[v] = m_1[v] + m_2[v] \\ \wedge\ e[v,w] = e_1[v,w] + e_2[v,w] \\ \wedge\ C(K, i, k, \overline{m}_1, \overline{e}_1) \wedge C(K, k, j, \overline{m}_2, \overline{e}_2) \end{array} \right)$$

The model-checking problem now turns to checking satisfiability of the formula $C(K, i, j, \overline{m}, \overline{e}) \wedge \neg(sim(\phi, i, j, \overline{m}, \overline{e}))$. Generated formulas are of size exponential in the chop-depth [10]. In case of negative polarity in ϕ, those formulas can be seen as quantifier-free fragments and checked by LinSAT decision algorithm which renders a doubly exponential algorithm. Otherwise, Presburger formulas are generated and decided by a triply exponential algorithm which leads to 4-fold exponential worst-case running time [10].

Regards to our model-checking example, the consistency condition for every trace between A and D ($C_{AD}(\overline{m}, \overline{e})$) is shown as follows:

$$\exists\, e_{AB},\ e_{AC},\ e_{BC},\ e_{BD},\ e_{CB},\ e_{CD}. \quad \begin{array}{llll} 1 & = m_A = & e_{AB} + e_{AC} \\ e_{AB} + e_{CB} & = m_B = & e_{BC} + e_{BD} \\ e_{AC} + e_{BC} & = m_C = & e_{CB} + e_{CD} \\ e_{BD} + e_{CB} & = m_D = & 1 \\ m_B > 0 & \Rightarrow\ e_{AB} > 0 & \end{array}$$

The DC formula is transformed to $m_B < 2$, so the model-checking problem for vertex A and vertex D is translated to:

$$\forall \overline{m}.\ (C_{AD}(\overline{m}, \overline{e}) \Rightarrow m_B < 2)$$

5.2 Simplification of Presburger formulas

Given a Kripke structure, the overall size of generated Presburger formulas is $O(u(|V| + |E|)|V|^c)$ [10]. Even for rather small model-checking problem, these formulas might become very large. And for the ease of model-checking algorithms, PA formulas contain a lot of redundant parts which could be removed before supplying for Cooper's algorithm or the Omega test. In this section, we analyze given PA formulas and derive a simplification process. The objective is compacting those PA formulas to be decided by other procedures later. Specifically, we target inputs for Cooper's algorithm, so specific aims of the simplification process are described below:

- **Reduce the number of quantifiers**. Cooper's procedure does quantifier alternation inside out; if quantifiers could be removed in a cheap way, it is really helpful for the algorithm later on.
- **Reduce coefficients of constraints**. We derive corresponding constraints with smaller coefficients to limit the chance of huge coefficients.
- **Reduce the number of A-Terms and B-Terms**. It is not obvious that removing inequalities will reduce the number of terms in the later recursive call. However, this is not the case with equations. As mentioned in Section 4.2, each equation hides an A-Term and a B-Term inside; therefore, eliminating equations is going to reduce the number of terms quite a lot.

5.2.1 Guarded Normal Form

Aiming towards a normal form which is convenient for cheap equation-based simplification, the Guarded Normal Form (GNF) is proposed [15]. An *implication guard* and a *conjunction guard* are proposition logic fragments in the following forms respectively:

$$\bigwedge_i L_i \Rightarrow \bigvee_j P_j$$

$$\bigwedge_i L_i \wedge \bigwedge_j P_j$$

We consider GNF in the Presburger setting where literals are comparison and divisibility constraints and guards consist of propositional connectives of quantified and quantifier-free formulas. First, GNF supports an easy way to express NNF [15]:

$$P \ \wedge \ Q \equiv \bigwedge_{i \in \phi} \wedge P \ \wedge \ Q$$

$$P \ \vee \ Q \equiv \bigwedge_{i \in \phi} \Rightarrow P \ \vee \ Q$$

$$\neg(\bigwedge_i L_i \Rightarrow \bigvee_j P_j) \equiv \bigwedge_i L_i \wedge \bigwedge_j \neg P_j$$

$$\neg(\bigwedge_i L_i \wedge \bigwedge_j P_j) \equiv \bigwedge_i L_i \Rightarrow \bigvee_j \neg P_j$$

As we can see from above equations, the *implication guard* and the *conjunction guard* are dual of each other. Transformation into NNF is done by simply pushing negation inside inner guarded formulas only. A formula is said to be in GNF if *conjunction guards* only occur in *implication guards* and vice versa. A *guarded formula* is put into GNF by applying equivalences:

$$\bigwedge_i L_i \Rightarrow (\bigwedge_k L'_k \Rightarrow \bigvee_l P'_l) \vee \bigvee_j P_j \equiv \bigwedge_i L_i \wedge \bigwedge_k L'_k \Rightarrow \bigvee_j P_j \vee \bigvee_l P'_l \tag{5.1}$$

$$\bigwedge_i L_i \wedge (\bigwedge_k L'_k \wedge \bigwedge_l P'_l) \wedge \bigwedge_j P_j \equiv \bigwedge_i L_i \wedge \bigwedge_k L'_k \wedge \bigwedge_j P_j \vee \bigwedge_l P'_l \tag{5.2}$$

Application of above rules helps to reduce the nesting of *guarded formulas*; in the next section, we study how to use GNF in connection with PA and simplify formulas as much as possible.

5.2.2 Equation-based simplification

Given a Presburger formula in the GNF, we sketch a simple algorithm to do cheap simplification on the formula. The first set of rules is for simplifying quantifier guards with equations inside:

$$\exists x.(nx = t \wedge \bigwedge_i L_i \Rightarrow \bigvee_j P_j) \equiv \top \tag{5.3}$$

$$\forall x.(nx = t \wedge \bigwedge_i L_i \wedge \bigwedge_j P_j) \equiv \bot \tag{5.4}$$

$$\exists x.(nx = t \wedge \bigwedge_i L_i \wedge \bigwedge_j P_j) \equiv n|t \wedge \bigwedge_i L_i[t/nx] \wedge \bigwedge_j P_j[t/nx] \tag{5.5}$$

$$\forall x.(nx = t \wedge \bigwedge_i L_i \Rightarrow \bigvee_j P_j) \equiv n|t \wedge \bigwedge_i L_i[t/nx] \Rightarrow \bigvee_j P_j[t/nx] \tag{5.6}$$

Rules 5.3 and 5.4 are preferable due to its strength of reducing a big formula to an obvious truth value. These truth values are possible to be propagated further to reduce even bigger formulas. Otherwise, rules 5.5 and 5.6 are also very helpful, because they are able to remove all equations in guards and eliminate corresponding quantifiers at the same time. A newly introduced divisibility constraint is an interesting point. If n is equal to 1, it is safely to remove; otherwise, that constraint is used to check the consistency of the term. Let b denote the constant and a denote the greatest common divisor of all coefficients of the term, we have the following condition:

$$n \mid at' + b \Longrightarrow gcd(n, a) \mid b$$

If the term does not satisfy the condition, the whole formula is reduced to a truth value again. One might think that equation-based reduction is helpful even without quantifiers involved. Certainly equations can be used to substitute variables so that substituted variables only occur in literals; however, coefficients may increase and the chance of introducing inconsistency in those formulas is not clear. Therefore, equation-based substitution without quantifiers is not considered here.

The second set of rules is for propagating independent literals to outer scopes. With a quantifier $Q \in \{\exists, \forall\}$, we have following relations:

$$Qx.(\bigwedge_{i1} L_{i1} \wedge \bigwedge_{i2} L_{i2} \Rightarrow \bigvee_j P_j) \equiv \bigwedge_{i1} L_{i2} \Rightarrow Qx.(\bigwedge_{i2} L_{i2} \Rightarrow \bigvee_j P_j) \tag{5.7}$$

$$Qx.(\bigwedge_{i1} L_{i1} \wedge \bigwedge_{i2} L_{i2} \wedge \bigwedge_j P_j) \equiv \bigwedge_{i1} L_{i1} \wedge Qx.(\bigwedge_{i2} L_{i2} \wedge \bigwedge_j P_j) \tag{5.8}$$

where $x \notin \bigwedge_{i1} L_{i1}$ and $x \in \bigwedge_{i2} L_{i2}$.

The advantage of applying these rules is two-fold. One is that quantified formulas become smaller and easier to be handled by decision procedures, another advantage is that independent literals come to outer scopes and hopefully are candidates for equation reduction rules (5.3-5.6).

Now our concern is how to use 5.3-5.8 effectively. The algorithm could reduce formulas outside in; however, if equations are not collected in the guards, we end up having many unexploited equations. Here we derive a recursive algorithm which executes in an inside-out order. We start from the deepest quantified formulas and propagate equations if they still occur as follows:

- Normalize the formula into GNF. Notice that the formula could be even reduce to a smaller one by evaluating it by the current guard, but we would think that is unnecessary. Because the smaller formula makes sure that variables occur in equation do not happen anywhere in the formula, simplification is done even before we know it is helpful or not. Therefore, the aim of this step is collecting equations into literals and keeping the structure small for doing traversal. One important point is that all constraints will be kept in the compact form so their coefficients are smallest.
- Apply 5.3-5.6 to reduce quantifiers and eliminate all equations in the current guard. This step may generate some independent constraints which are candidates for the next step.
- Partition literals by whether they consist of quantifiers or not. Take independent literals out of current quantifiers, which causes the formula not in GNF anymore. The bigger formula will be normalized into GNF recursively in the next iteration.

One decision has to be made is whether we should push quantifiers as down as possible. The process is based on the following rules:

$$\exists x.(\bigwedge_i L_i \Rightarrow \bigvee_j P_j) \equiv \bigwedge_i \forall x.L_i \Rightarrow \bigvee_j \exists x.P_j \tag{5.9}$$

$$\forall x.(\bigwedge_i L_i \wedge \bigwedge_j P_j) \equiv \bigwedge_i \forall x.L_i \wedge \bigwedge_j \forall x.P_j \tag{5.10}$$

These rules result in quantified formulas in the smallest size. If decision procedures are able to recognize these patterns, it is unnecessary to do so. We are going to come back to this simplification process later in the report.

5.3 Summary

We have presented side-condition Presburger formulas and how they are generated from the model checker. Due to high complexity of these formulas, we derive a simplification algorithm to reduce them as much as possible. Reduction is based on understanding structures of these formulas and establishing logic rules. Hopefully, simplified formulas are small enough to be solved by decision procedures in a reasonable amount of time.

CHAPTER 6

Experiments on Presburger Arithmetic

This chapter is organized as a series of experiments on different aspects of Presburger Arithmetic. The chapter starts with discussion of generating various Presburger fragments. These formulas are used as inputs for testing different algorithms later on. The next part deals with simplification of Presburger formulas; it pops up from the fact that side-condition Presburger fragments are complex and able to be reduced by some cheap quantifier elimination. After that, we discuss design and implementation of Cooper's algorithm in a sequential manner. Some design choices are made which have influence on both sequential and parallel versions, and we attempt to do a benchmark to decide which option is good for the procedure.

6.1 Generation of Presburger fragments

Our main source of Presburger formulas is from the model checker of Duration Calculus. However, as discussed in Chapter 5, generated Presburger fragments are in a huge size even for a very small model-checking problem, and testing and optimizing decision procedures on those formulas are pretty difficult. Also there is no common benchmark suite for Presburger formulas; therefore, we attempt

to generate test formulas which are controllable in terms of size and complexity and postpone working on the real-world formulas until a later phase. Our test formulas are from two sources:

- *Hand-annotated Presburger formulas*: in most of the cases, these formulas are able to be quickly solved by hand. They serve the purpose of ensuring the correctness of decision procedures and testing some facets of Presburger formulas which are easier to be constructed by hand.
- *Automatically-generated Presburger formulas*: we generate these formulas by formulation of Pigeon Hole Principle, and the detailed procedure is presented later in this section. We use this formulation to generate formulas whose sizes are controllable and satisfiability is predetermined. These formulas allow us to test with controllably big inputs which are particularly helpful in context of parallel execution.

Here we describe our formulation of Pigeon Hole Principle: *given N pigeons and K holes, if $N \leq K$ there exists a way to assign the pigeons to the holes where no hole has more than one pigeon; otherwise, no such assignment exists.* Certainly, there are many ways to formulate Pigeon Hole Principle in logic, here we employ a very simple interpretation.

Let x_{ik} be the predicate where pigeon i is in hole k. The predicate F shows that if pigeon i is in the hole k, there is no other pigeon in that hole:

$$\texttt{F(i, k)} = \texttt{x}_{\texttt{ik}} \Rightarrow \bigwedge_{\substack{1 \leq j \leq N \\ j \neq i}} \neg \texttt{x}_{\texttt{jk}}$$

The predicate G shows that each pigeon is assigned one hole:

$$\texttt{G(i)} = \bigvee_{1 \leq \texttt{k} \leq \texttt{K}} \texttt{x}_{\texttt{ik}}$$

The predicate H shows that each pigeon is in only one hole:

$$\texttt{H(i, k)} = \texttt{x}_{\texttt{ik}} \Rightarrow \bigwedge_{\substack{1 \leq l \leq K \\ l \neq k}} \neg \texttt{x}_{\texttt{il}}$$

The principle is formulated as follows:

$$\texttt{Pigeon(N, K)} = \bigwedge_{\substack{1 \leq i \leq N \\ 1 \leq k \leq K}} \texttt{F(i, k)} \wedge \bigwedge_{1 \leq \texttt{i} \leq \texttt{N}} \texttt{G(i)} \wedge \bigwedge_{\substack{1 \leq i \leq N \\ 1 \leq k \leq K}} \texttt{H(i, k)}$$

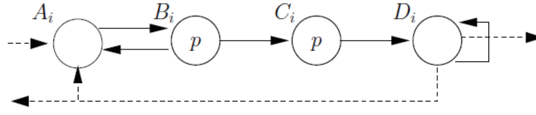


Figure 6.1: A N-sequence Kripke structure [15].

According to the principle, satisfiability of Pigeon(i, k) is easy determined by values of N and K. And size of these predicates can easily scaled by the big values of arguments which are really helpful for testing and optimizing parallel algorithms. Up till now we examine the Pigeon Hole Principle in a propositional setting, to inject linear constraints and quantifiers we consider the following encodings:

$x_{ik} \leftarrow \top$	$x_{ik} \geq 0$	$x_{ik} = 0$	$2 \mid x_{ik}$
$x_{ik} \leftarrow \bot$	$x_{ik} < 0$	$x_{ik} \neq 0$	$\neg(2 \mid x_{ik})$

and define quantified formulas in the form of $\exists x_{11}, \ldots, x_{NK}.$ Pigeon(N, K). We call these formulas *Pigeon-Hole* Presburger formulas to attribute their origins. So these formulas contain $N * K$ variables and $N * K$ quantifiers; we can omit some quantifiers to adjust the number of quantifiers by our needs. Different encodings give us different preferences of constraints which in turn demonstrate some interesting aspects in decision procedures. This procedure of generating Presburger formulas is easy to extend by using other encodings and there are many other ways to formulate the principle in logic, resulting in various subsets of Presburger formulas. We are open for extending the procedure for new subsets of Presburger fragments in the future. Detailed source code of the formulation can be found in Appendix B.4.

For Presburger fragments arise from the model checker, we consider Kripke structures by concatenating N identical automata which is illustrated in Figure 6.1. The DC formula is $\Box(\mathtt{l} < 5 \Rightarrow \int \mathtt{p} < 3)$ where $\Box\phi$ is defined by $\neg(\mathtt{true}^\frown(\neg\phi)^\frown\mathtt{true})$. Resulting Presburger fragments are quite big and they are used as the input for the simplification process which is discussed in the next section.

6.2 Simplification of Presburger formulas

In Section 5.2, we have discussed an algorithm to quickly simplify Presburger fragments. This section presents experimental results of the algorithm and compares them with previous approaches. Our experiment is conducted as follows:

- Four side-condition Presburger formulas are generated from 2-sequence, 3-sequence, 4-sequence and 5-sequence automata with the associated DC formula(see 6.1 for details).
- Simplified formulas are recorded by running a simplification process by Hansen *et al.* (HB's) [15] and one by our algorithm (Ours).
- Simplified formulas are fed into the SMT-solver Z3 for quantifier elimination and evaluation.

Our experimental results are summarized in Table 6.1. These results show that our simplification process at least reduces 4.5% of number of quantifiers more than the other method, and the deepest nesting of quantifiers is also 10.5% smaller. And it is also clear that less complex formulas are easier for Z3 to solve. This brings the hope that recognizing patterns of some Presburger fragments and quickly simplifying them is important to assist decision procedures. By performing this experiment, we demonstrate that further simplification of Presburger fragments is possible and meaningful to reduce stress on decision procedures. We are going to incorporate more elements to simplify formulas as much as possible before solving them.

	No. of quantifiers (Original/HB's/Ours)	Deepest nesting of quantifiers (Original/HB's/Ours)	Solving time (HB's/Ours)
2-seq	3190/729/583	54/25/19	0.31/0.28 s
3-seq	10955/2823/2352	83/39/30	0.9/0.78 s
4-seq	26705/7638/5165	113/54/42	2.27/1.47 s
5-seq	53284/16830/14501	144/70/55	56.47/6.32 s

Table 6.1: Experimental results of two simplification algorithms.

6.3 Optimization of Cooper's algorithm

Before going to implement a parallel variant of Cooper's algorithm, we consider some design choices for the algorithm itself. These design choices affect both

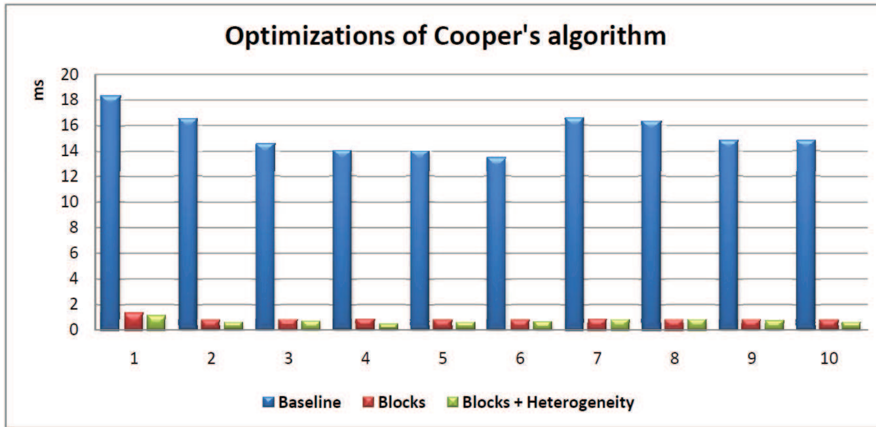


Figure 6.2: Some optimizations on Cooper's algorithm.

sequential and parallel versions, but to keep things simple we justify the selection by a benchmark of the sequential algorithm only. There are some improvements which have been done to contribute to a better implementation of Cooper's procedure:

- **Reduce all literals as much as possible**. By reducing, we mean all constraints are evaluated to truth values whenever possible, and their coefficients are reduced to the smaller values. This process will lead to early evaluation of some parts of a formula, and huge coefficients are avoided as much as possible.
- **Keep a formula in a uniform way**. This improvement relies on our experience of GNF, and uniform formulas not only have small depths of recursion but also collect many formulas in their collection and introduce more chances of concurrency, which is important for the purpose of parallel execution later on.

The test set consists of ten hand-annotated Presburger formulas, each one has 3-4 nested universal quantifiers and a disjunction of 3-4 equalities. The benchmark is performed in the following manner: each test is run 100 times and execution time is calculated by average execution time of 100 sessions. Moreover, some sessions were executed before the benchmark, so every function is ensured in the warm state in the time of benchmarking.

Here we consider two important optimizations which have been mentioned in Section 4.2: *eliminating blocks of quantifiers* and *using heterogeneous coeffi-*

cients of literals. The idea of eliminating blocks of quantifiers is clear, instead of manipulating a huge formula, we split the work into many small pieces and manipulate many small formulas. As illustrated in Figure 6.2, pushing blocks of quantifiers into small formulas results in $10 - 20\times$ performance gain compared to the baseline version. The result is significant because smaller formulas easily fit into caches, so manipulating them is much faster in practice. Using heterogeneous coefficients adds up a slight performance gain as demonstrated in the figure. Their advantage is two-fold; we save one traversal through the formula to collect all coefficients so resulting coefficients are smaller and resulting quantifier-free formulas are easier to evaluate in the later phase. Next section discusses a new perspective of Cooper's algorithm regarding parallel execution.

6.4 Summary

In this chapter, we have presented various experiments on Presburger Arithmetic. Generation of different Presburger fragments is important for the testing purpose, and simplification is helpful for reducing stress on decision procedures. Optimization of Cooper's algorithm is paving the way to derive better parallel algorithms. We discuss various aspects of parallelism on decision procedures in the next chapter.

Chapter 7

Parallel execution of decision procedures

In this chapter, parallel versions of Cooper's algorithm and the Omega Test are discussed with many details regarding their concurrency, efficiency and scalability. While Cooper's algorithm is presented in a complete procedure, the Omega Test is used as a partial process assisting simplification of complex Presburger formulas. In each section, some test sets are selected for experimentation and experimental results are interpreted. These test sets are carefully chosen from sources of Presburger fragments in Section 6.1 and experiments if not explicitly mentioned are performed on the 8-core 2.40GHz Intel Xeon workstation with 8GB shared physical memory. Each test is chosen so that its execution time is smaller than 12 seconds. Measurement of execution time is done in the following manner:

- Benchmark functions are executed in a low-load environment.
- Some sessions are performed several times so that every function is a warm state before benchmarking.
- Each test is run `10` times and average execution time is recorded.

7.1 A parallel version of Cooper's algorithm

In this section, we discuss parallelism in Cooper's algorithm and sketch some ideas about how to exploit it. We come up with design and implementation for the procedure and present experimental results.

7.1.1 Finding concurrency

As discussed in Section 6.3, eliminating blocks of quantifiers is good because many small formulas are faster to manipulate. Other than that there is a chance of concurrency in distributing blocks of quantifiers, where small formulas are independent of each other and it is possible to eliminate quantifiers in parallel. Our idea of parallel elimination is based on following rules where formulas are in NNF:

$$\exists\ x_1 ... x_n . \bigvee_i F_i \equiv \bigvee_i \exists\ x_1 ... x_n . F_i \tag{7.1}$$

$$\forall\ x_1 ... x_n . \bigwedge_i F_i \equiv \bigwedge_i \forall\ x_1 ... x_n . F_i \tag{7.2}$$

However, for arbitrary NNF formulas, the degree of parallelism may be limited because of small disjunctions (or conjunctions). One way to enhance concurrency is converting formulas into DNF. Normally a DNF formula is in a form of a huge disjunction of many inner conjunctions and quantifier elimination can be distributed to inner conjunctions immediately. Because conversion into DNF causes the formula's size to grow very fast, we only do DNF conversion once at the outermost level of quantifiers.

Because there are still some cases where DNF formulas do not expose enough concurrency, we seek concurrency inside the procedure. First, due to recursive nature of Presburger formulas, these formulas are represented by tree data structures. Certainly we can manipulate these formulas by doing operations on tree in a parallel manner, for example, doing parallel evaluation of tree branches. Second, certain parts in quantifier elimination could be done in parallel. As can be seen from Figure 7.1, `Get Coefficients`, `Get A-Terms` and `Get B-Terms` have no order of execution. We are able to create three **Tasks** to run them concurrently. Similarly `Least Satisfying Assignment` and `Small Satisfying Assignments` could be calculated in parallel. The figure preserves relative ratios between sizes of different tasks where `Small Satisfying Assignments` is the largest task (it consists of |`B`| (or |`A`|) times substitution of a variable by a term in the formula) and `Eliminate Variable` is a very lightweight task where

we assemble different results. Assuming that the number of B-Terms are always smaller than that of A-Terms, we have a rough estimation of parallelism based on the *DAG model of multithreading* as follows:

- Assume that `Eliminate Variable` is an insignificant task and omit it.
- Estimate each task by the number of traversals through the whole Presburger formula.
- *Work* $= 1 + 1 + 1 + 1 + 1 + |B| = |B| + 5$
- *Span* $= 1 + 1 + 1 = 3$
- *Parallelism Factor* $= (|B| + 5)/3$

Roughly speaking, *Parallelism Factor* is bounded by $(|B| + 5)/3$, so we can hardly achieve a good speedup if the number of B-Terms (or A-Terms) is too small. Actually the number of terms is quite small due to symbolic representation of big disjuncts and the choice of the smallest number of terms in projection which leads to limited concurrency in each elimination step. One interesting thing is after each elimination step, big disjuncts consists of $1 + |B|$ inner formulas which are subject to parallel elimination. Again good concurrency requires a big number of terms.

7.1.2 Design and Implementation

The algorithm we use follows ideas of Cooper's algorithm discussed in Section 4.2 and there are some differences related to eliminating blocks of quantifiers and using heterogeneous coefficients as discussed in Section 6.3. Here we focus on data representation which will be shown to have a big influence on performance later on.

The type *Term* is employed to represent a linear term $c + \Sigma_i a_i x_i$:

- It consists of a constant c and a collection of variables x_i and corresponding coefficients a_i.
- Variables cannot be duplicated, and each variable has exactly one coefficient.
- Arithmetic operations such as addition, subtraction between *Terms*, unary minus of *Terms*; multiplication and division by a constant are also supported.

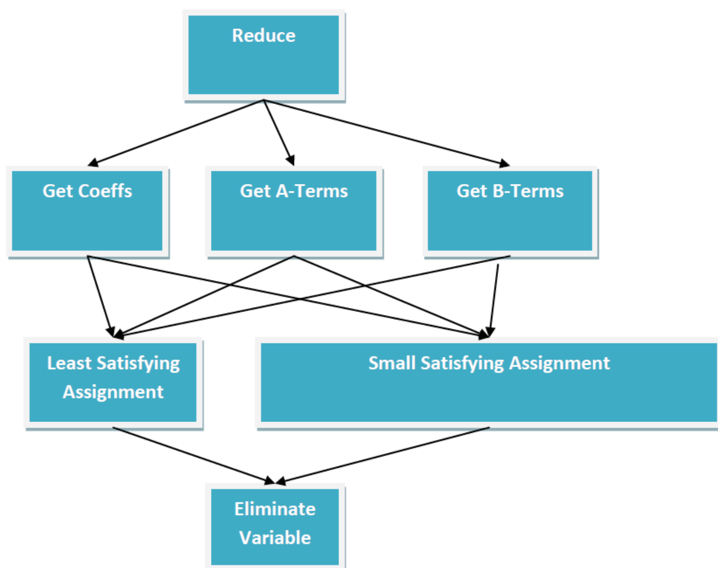


Figure 7.1: Task graph of an elimination step in Cooper's algorithm.

A natural representation of *Term* in F# is the following type:

```
type Term = struct
    val Vars: Map<string, int>
    val Const: int
end
```

We intend to use *Term* as a `struct` because it is a value type stored in stacks and garbage collectors do not have to take care of them. However, in F# a `Map` is implemented in a form of a balanced binary tree, each operation of adding and removing elements results in inserting and deleting nodes in a binary tree. Therefore, if arithmetic operations on *Term* are used often, a lot of nodes are created and discarded so garbage collectors have to work quite often. We devise another representation of *Term* as follows:

```
type Term = struct
    val constant: int
    val coeffs: int []
    val vars: string []
end
```

We make use of arrays of primitive types instead of `Map`, certainly it is more

difficult to ensure no duplication in arrays but we have a more compact representation and avoid small object allocation and deallocation. One interesting thing is the separation of coefficients and variables is more efficient when we need to update coefficients, multiply or divide *Term* by a constant because we only change `coeffs` array and share other unchanged fields. Here we use this representation for our experiments.

The type *Formula* represents Presburger formulas with following properties:

- It is a recursive datatype to support the recursive nature of Presburger formulas.
- It has elements to accommodate the symbolic interpretation of big disjuncts.
- Presburger formulas are in a compact form by flattening formulas and employing collection-based interpretation.

We have *Formula* in F# as follows:

```
type Formula =
    | TT
    | FF
    | C of Term * CompType
    | D of int * Term
    | Not of Formula
    | And of Formula list
    | Or of Formula list
    | SAnd of Formula * (string * int) list
    | SOr of Formula * (string * int) list
    | E of string list * Formula
    | A of string list * Formula
```

In this representation, the depth of formulas is easy to be reduced by flattening formulas. *Formula* can be seen as a tree datatype with different numbers of branches at each node. Some ideas of parallel execution on this tree can be summarized as follows:

- Parallel execution is done until a certain depth.
- Size of formulas (number of literals) can be used to divide tasks running in parallel.
- An operation is done sequentially if the size of a branch is too small for parallelism.

We have done an experiment with parallel execution on a binary tree and present results in the next section.

7.1.3 Experimental Results

We have a version of binary tree and a parallel `map` function and we attempt to measure speedups on different `oWorkload` functions:

```
type BinTree =
  | Leaf of int
  | Node of BinTree * BinTree

let pmap depth oWorkload tree =
    let rec pmapUtil t d =
        match t with
        | Leaf(n) -> if oWorkload(n) then Leaf(1) else Leaf(0)
        | _ when d = 0 -> map oWorkload t
        | Node(t1, t2) ->
            let t1' = Task.Factory.StartNew( fun() -> pmapUtil t1 (d-1))
            let t2' = Task.Factory.StartNew( fun() -> pmapUtil t2 (d-1))
            Task.WaitAll(t1', t2')
            Node(t1'.Result, t2'.Result)
    pmapUtil oWorkload tree depth
```

Workload functions are simple for-loops running a specified number of times:

```
// Workload functions
val oConstant : int -> bool
val oLog : int -> bool
val oLinear : int -> bool
```

Figure 7.2 shows that speedup factors are really bad when workloads are not heavy enough and a good speedup of 6× is obtained with a rather big workload. Consider our scenario of manipulating Presburger formulas, trees are of big size but tasks at each node are tiny, and at most we manipulate small *Terms* by their arithmetic operations. Therefore, it hardly pays off if we try to parallelize every operation on *Formula*.

The experiment with Cooper elimination is performed on 10 *Pigeon-Hole* Presburger formulas (see Section 6.1 for their construction), and each of them consists of 32 disjunctions which can be resolved in parallel. Their properties are summarized in Table 7.1.

Figure 7.2: Speedup factors with different workloads.

Test No.	Pigeons	Holes	Variables	Quantifiers	Literals
1	21	1	21	3	483
2	22	1	22	3	528
3	23	1	23	3	575
4	24	1	24	3	624
5	25	1	25	3	675
6	26	1	26	3	728
7	27	1	27	3	783
8	28	1	28	3	840
9	29	1	29	3	899
10	30	1	30	3	960

Table 7.1: Test set for Cooper elimination.

Each formula contains 32 smaller formulas which can be resolved in parallel. Experimental results are presented in Figure 7.3. As can be seen from the figure, speedup factors are around $4 - 5\times$; these suboptimal speedups can be explained by a fast-growing number of cache misses when the problem size increases. Some other reasons of suboptimal speedups could be found in Section 3.2.

Cooper evaluation is done with some *Pigeon-Hole* Presburger formulas. They are chosen in a way that their truth values are false, so it ensures that the algorithm has to iterate through the whole search space to search for an answer.

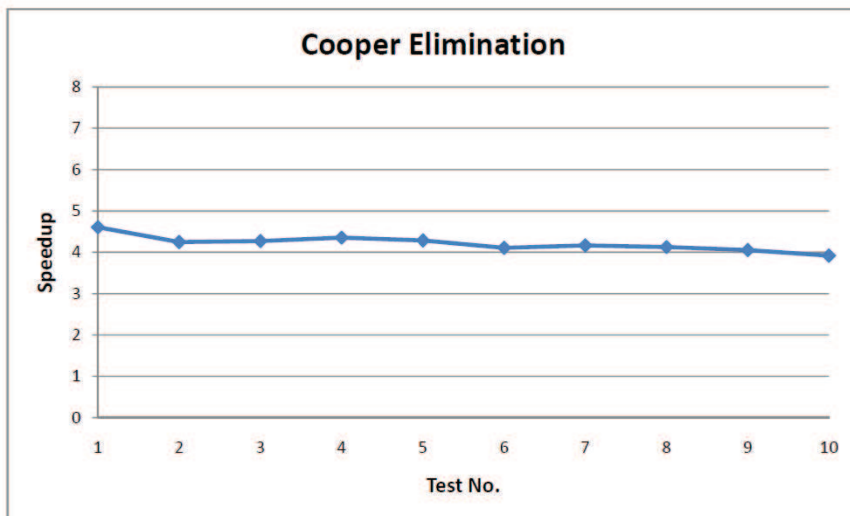


Figure 7.3: Speedup factors in Cooper elimination only.

Some characteristics of tests formulas are presented in Table 7.2.

Test No.	Pigeons	Holes	Variables	Quantifiers	Literals
1	4	2	8	8	56
2	8	1	8	8	80
3	9	1	9	9	99
4	5	2	10	10	80
5	10	1	10	10	120
6	11	1	11	11	143
7	4	3	12	12	96
8	6	2	12	12	108
9	12	1	12	12	168
10	13	1	13	13	195

Table 7.2: Test set for Cooper evaluation.

Experimental results are shown in Figure 7.4. Speedup factors are good, ranging from 5x to 8x, because search spaces are huge and the algorithm has been traversed through them to validate results. Detailed source code of Cooper elimination and evaluation can be found in Appendix B.5.

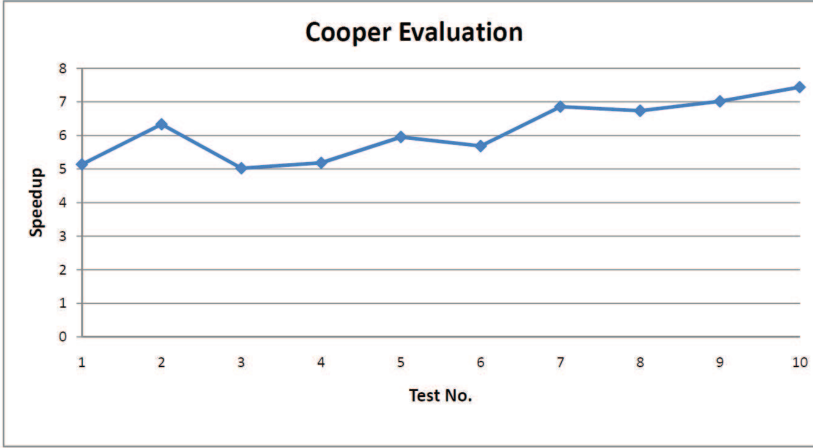


Figure 7.4: Speedup factors in Cooper evaluation.

7.2 A parallel version of the Omega Test

This section is about our concern of parallelism in the Omega Test. The section starts with discussion about concurrency occurring in the algorithm and ideas about how to exploit it. Thereafter we present our design, implementation and experimental results.

7.2.1 Finding concurrency

As mentioned in Section 4.2.2, converting Presburger formulas into DNF is inevitable for the Omega Test. DNF is a significant bottleneck of the algorithm because formulas may grow exponentially. Take a formula $\bigwedge_{\mathtt{i}=1}^{\mathtt{m}} \mathtt{L_i} \wedge \bigwedge_{\mathtt{j}=1}^{\mathtt{n}} (\mathtt{L1_j} \vee \mathtt{L2_j})$ as an example, its DNF version is a disjunction of $2^{\mathtt{n}}$ conjunctions of $\mathtt{m} + \mathtt{n}$ literals. DNF is a challenge for employing the Omega Test, but there is a chance of massive concurrency in this procedure. In the above example, $2^{\mathtt{n}}$ conjunctions are totally independent and ready for being resolved in parallel. To do so we rely on the following rule (which is similar to Equation 7.2):

$$\exists\, x_1 ... x_n . \bigvee_i \bigwedge_j L_{ij} \equiv \bigvee_i \exists\, x_1 ... x_n . \bigwedge_j L_{ij} \tag{7.3}$$

Using DNF, we can easily divide a formula into many smaller ones and apply the Omega Test independently. The parallel pattern used here is *Data Parallelism*, when we have the same operation working on a big collection of data. As

discussed in Section 2.2, F# provide very good constructs for data parallelism such as `Parallel.For(Each)`, `Array.Parallel` and `PSeq` module.

Looking into details of the Omega Test procedure, it is easy to recognize that their three basic blocks could be executed in parallel because there is no requirement for order of executions among *Real Shadow*, *Dark Shadow* and *Gray Shadow*. We can easily spawn three independent `Tasks`, but there is a little coordination which should be done between these tasks. First of all, if *Real Shadow*'s task completes first and returns false, the other tasks should be terminated immediately. Similarly, if *Dark Shadow*'s task returns true, the other running tasks should be stopped right away. In the worst-case scenario, the result is returned from *Gray Shadow*'s task. It is likely that this task is the longest running one so total execution time is comparable to execution time of this task. This idea of three parallel coordinated tasks can be easily implemented in F# using `Task` and `CancellationTokenSource` constructs.

We have sketched our ideas of parallelizing Omega Test. These two ideas can be incorporated into a full procedure with a sensible way of controlling the degree of parallelism. We recognize that exponential growth of formulas' size is the biggest hindrance, especially *Gray Shadow* happens quite often and causes formulas to be normalized into DNF again and again. Noticing that consistency predicates (see Section 5.1) consist of literals with coefficients zero or one, we might think that *Exact Shadow* is a good choice for quickly resolving formulas of this form. The original form of *Exact Shadow* (see Equation 4.1) is adapted in a form of strict comparisons:

$$\exists \mathtt{x}.\ \beta < \mathtt{bz} \land \mathtt{cz} < \gamma \iff \mathtt{c}\beta + \mathtt{1} < \mathtt{b}\gamma \quad (7.4)$$

Basically, our parallel procedure works with a lot of small conjunctions and with each conjunction, it does *Exact Shadow* to cheaply eliminate quantifiers. We know that this approach is incomplete because after some substitution coefficients of literals increase and the procedure has no way to go ahead. But given the fact that the consistency predicates are heavily used in side-condition Presburger fragments, even removing a fraction of quantifiers is really helpful. We are going to introduce the detailed procedure in the next section.

7.2.2 Design and Implementation

We sketch the algorithm of using Omega Test in a general case, a specific parallel algorithm can be derived by our discussion of **Finding Concurrency** above:

- The Presburger formula is converted into DNF.

- For each conjunction of literals in the new formula, quantifiers are eliminated recursively by the following procedure:
  - Suppose a formula is in a form of $\exists \mathtt{x}.\ \phi(\mathtt{x})$ and independent literals which do not contain `x` are stored separately.
  - If `x` is unbounded (literals having `x` only consist of less-than or greater-than comparisons), the quantifier `x` is safely removed. The new conjunction contains only independent literals.
  - For bounded literals, they are categorized into two opposite groups of $\beta < \mathtt{bx}$ and $\mathtt{cx} < \gamma$. For each pair of opposite literals, if they satisfy the condition of *Exact Shadow* (b=1 or c=1) we apply the shadow to form a new literal; otherwise, the procedure stops immediately.
  - The new conjunction consists of independent literals and newly-formed literals, the quantifier `x` is safely removed. We eliminate the next quantifier by running the same procedure on the new conjunction.
- The quantifier elimination procedure completes when no quantifier is left or there are some quantifiers but coefficients of literals are different from one.

We analyze the algorithm in a case of our specific input. We have a quantified formula with `k` quantifiers and a conjunctions of `m` literals and `n` disjunctions of two literals. As discussed above, the corresponding formula in DNF consists of $2^{\mathtt{n}}$ conjunctions of $(\mathtt{m} + \mathtt{n})$ literals. Examining each conjunction separately, each quantifier elimination step causes a product between two lists of size $(m + n)$. Therefore, we have a worst-case complexity of $\Theta((\mathtt{m} + \mathtt{n})^{2\mathtt{k}})$. Certainly, this does not happen often in practice, if one of these lists is empty we have size of new conjunctions remaining linear of $\mathtt{m} + \mathtt{n}$. The worst-case complexity of the whole procedure is $\mathtt{Work} = \Theta(2^{\mathtt{n}}(\mathtt{m} + \mathtt{n})^{2\mathtt{k}})$. Analyzing the algorithm by the *DAG model of multithreading*, $\mathtt{Span} = \Theta((\mathtt{m} + \mathtt{n})^{2\mathtt{k}})$, and we have *Parallelism Factor* of $\Theta(2^{\mathtt{n}})$. We can say this one is an instance of embarrassingly-parallel algorithms, and its parallelism is only bounded by the number of used processors.

Regarding specific implementation details, we have written two different versions of the algorithm: one is **List-based** and one is **Array-based**. We want to examine performance of `List` and `Array` in the context of parallel execution although it requires a little effort to translate from **List-based** version to **Array-based** one. Certainly, with each version we implement a parallel variant and a sequential one. They are running on the same codebase, there are only some small differences regarding parallelism constructs. In the next section, we present our experiment on a small test set and some conclusions about the algorithm and its parallel execution.

7.2.3 Experimental Results

We create a testset from consistency predicates of N-sequence automata in Figure 6.1. For each automaton in 3-sequence, 4-sequence and 5-sequence automata, we select five arbitrary consistency predicates. Therefore, we have a testset of 15 Presburger formulas ordered by their number of quantifiers and their size. We notice that each consistency predicates is a conjunction between literals and disjunctions which in turn contain two literals, characteristics of the testset are summarized in Table 7.3.

Automaton	**Quantifiers**	**Literals**	**Disjunctions**
3-seq	3	8	5
4-seq	6	13	7
5-seq	10	19	9

Table 7.3: Test set for the Omega Test.

From the table we can see that at most the procedure resolves 2^5, 2^7 or 2^9 Omega Tests in parallel. Because these numbers are usually bigger than the number of hardware threads, using some work balancing mechanisms would benefit the whole procedure. In the **List-based** version, work balancing is done by *PLINQ* engine without users' intervention. In our **Array-based** variant, we have more control over distribution of work on different cores. Here we use a partitioning technique which divides work into balanced chunks for the executing operation.

We have Figure 7.5 indicate speedups of different parallel versions. As we can see, speedup factors seem to decrease when the input size increases. This trend can be explained by the number of cache misses eventually grows when the input becomes bigger. However, speedup factors are quite good for both versions, they are around 5× in the worst cases. The **Array-based** approach is always more scalable than **List-based** one and this happens not only because we have better work balancing in **Array-based** variant but also because locality of references is ensured by keeping data closely in the array-based representation. Our **Array-based** approach performs quite well in the whole testset, speedup factors are around 6× even for big inputs. The **Array-based** implementation is not only more scalable but also surpasses its competitor in parallel execution. Figure 7.6 shows relative speedups between two parallel variants; their performance gap seems to be bigger when the problem size increases.

The results of our experiment are summarized in Table 7.4. Our cheap quantifier elimination based on *Exact Shadow* is very effective; it is able to eliminate all quantifiers for consistency predicates from 3-sequence and 4-sequence automata.

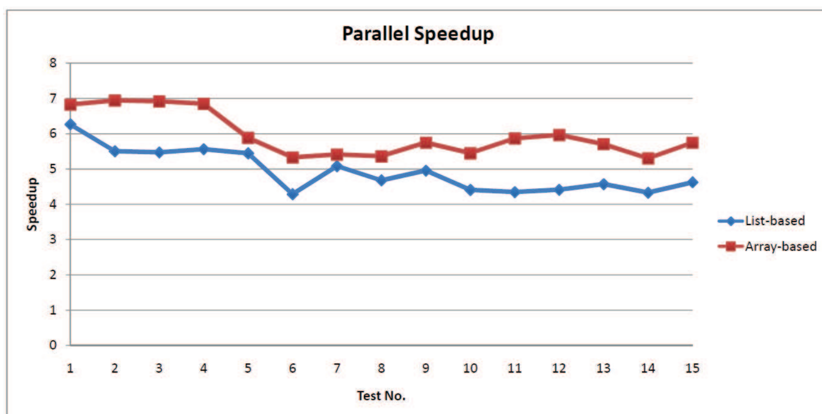


Figure 7.5: Parallel speedups of different approaches.

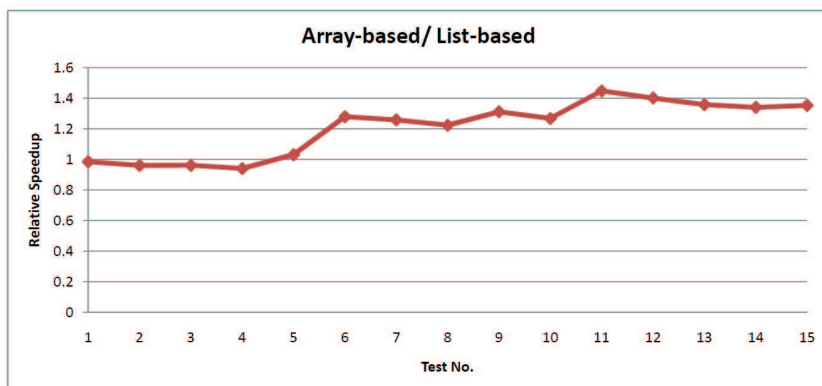


Figure 7.6: Relative speedups between **Array-based** approach and **List-based** approach.

In the case of 5-sequence automaton, the procedure stops with three quantifiers left, we think that even this partial elimination could help to simplify Presburger formulas. Detailed source code of the Omega Test can be found in Appendix B.6 and B.7.

Automaton	Quantifiers before	Quantifiers after
3-seq	3	0
4-seq	6	0
5-seq	10	3

Table 7.4: Results of the Omega Test.

7.3 Summary

We have discussed parallelism aspects of Cooper's algorithm and the Omega Test. Each discussion follows with some experiments, reasonable speedups have been obtained. It is easy to come up with a correct implementation of these parallel algorithms; however, it is not free to achieve good efficiency. The method we use for implementation has some interesting points:

- We employ a trial-and-error process. Each attempt is evaluated by profiling applications with some specific inputs.
- Some low-level tools are used including **CLR Profiler** and **ILSpy**. The former provides more information regarding memory and garbage collection and the latter helps to translate F# to other .NET languages for understanding low-level implementation and optimizing F# code.
- We follow the idiom of little memory allocation and little garbage collection; therefore, code is optimized so that their memory footprints are small and unnecessary garbage collection is avoided.

F# has provided a very convenient environment for parallel programming. We can start prototyping algorithms without worrying about sophisticated synchronization constructs and F# interpreter has been a good indicator for parallelism, so that we can decide when to start benchmarking. However, it is still difficult to optimize algorithms for parallelism because memory footprints of F# programs are quite large and garbage collection has a bad influence on performance in an unpredictable way.

CHAPTER 8

Conclusions

In this work, we have studied multicore parallelism by working on some small examples in F#. The result is promising; the functional paradigm makes parallel processing really easy to start with. We have also studied theory of Presburger Arithmetic, especially their decision procedures. We have applied the functional paradigm in a simplification algorithm for Presburger formulas and two important decision procedures: Cooper's algorithm and the Omega Test. Actually the paradigm is helpful because formulation in a functional programming language is really close to problems' specification in logic and the like. The outcome is clear; we can prototype these decision algorithms in a short time period and experiment with various design choices of parallelism in a convenient way. Some concrete results of this work are summarized as follows:

- Propose a new simplification algorithm which reduces 4.5% of numbers of quantifiers and 10.5% of nesting levels of quantifiers compared to the old method.
- Show that *eliminating blocks of quantifiers* and *using heterogeneous coefficients of literals* are helpful, which contribute up to $20\times$ performance gain compared to the baseline implementation.
- Present parallelism concerns in Cooper's algorithm. We have got $4-5\times$ speedup in elimination of formulas consisting of 32 disjunctions. The

evaluation phase is easier to parallelize, and we have achieved $5-8\times$ speedup for the test set of *Pigeon-Hole* Presburger formulas.

- Explore the chance of parallelism in the Omega Test. We have implemented a partial process using *Exact Shadow*, and have got $5-7\times$ speedup for the test set of consistency predicates.

While doing the thesis, we have learned many interesting things related to functional programming, parallelism and decision procedures:

- Functional programming and multicore parallelism is a good combination. We can prototype and experiment on parallel algorithms with little effort thanks to no side effect and a declarative way of thinking.
- The cache is playing an important role in multicore parallelism. To optimize functional programs for parallel efficiency, reducing memory allocation and deriving good cache-locality algorithms are essential. For example, our experience shows that using *Array-based* representation might be a good idea to preserve cache locality, which then contributes to better speedups.
- Cooper's algorithm and the Omega Test are efficient algorithms for deciding Presburger formulas, and we have discovered several ways of parallelizing these algorithms and achieved good speedups in some test sets. However, memory seems to be an inherent problem of the algorithms and parallelization is not the way to solve it.

We have done various experiments on Presburger Arithmetic from generating, simplifying to deciding those formulas. However, we have not done them in a whole process for our Presburger formulas of interest. We have the following things for our future work:

- Simplify Presburger formulas further, and aim at space reduction particularly for formulas generated by the model checker.
- Combine Cooper's algorithm and the Omega Test so that we can derive a decision procedure with good characteristics of the two algorithms.
- Optimize Cooper's algorithm and the Omega Test in terms of parallel execution. Especially, consider optimizing data representation for better cache locality and smaller memory footprints.
- Use solving of divisibility constraints in evaluation to reduce search spaces which are quite huge, especially with big coefficients and a large number of quantifier eliminations.

References

[1] Nikolaj Bjørner. Linear Quantifier Elimination as an Abstract Decision Procedure. In *IJCAR*, 2010.

[2] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1*, 1975.

[3] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 1996.

[4] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08. Society for Industrial and Applied Mathematics, 2008.

[5] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[6] D. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, 1972.

[7] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. *Electr. Notes Theor. Comput. Sci.*, 2005.

[8] Klaedtke Felix. Bounds on the automata size for Presburger arithmetic. *ACM Trans. Comput. Logic*, 2008.

[9] Michael J. Fischer and Michael O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, 1974.

[10] Martin Fränzle and Michael R. Hansen. Efficient Model Checking for Duration Calculus. *Int. J. Software and Informatics*, 2009.

[11] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 1988.

[12] Tuukka Haapasalo. Multicore Programming: Implicit Parallelism. `http://www.cs.hut.fi/~tlilja/multicore`, 2009.

[13] Youssef Hamadi and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT*, 2009.

[14] Michael R. Hansen. Duration Calculus. Technical report, Informatics and Mathematical Modelling Technical University of Denmark, 2010.

[15] Michael R. Hansen and Aske Wiid Brekling. On Tool Support for Duration Calculus on the basis of Presburger Arithmetic. In *TIME 2011 (to be appeared)*, 2011.

[16] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, July 2008.

[17] Predrag Janicic, Ian Green, and Alan Bundy. A comparison of decision procedures in Presburger arithmetic. In *University of Novi Sad*, 1997.

[18] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer Publishing Company, Incorporated, 2008.

[19] Chris Lyon. Server, Workstation and Concurrent GC. *MSDN*, 2004.

[20] David C.J. Matthews and Makarius Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, DAMP '10. ACM, 2010.

[21] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, 2004.

[22] Michael Norrish. Complete integer decision procedures as derived rules in HOL. In *Theorem Proving in Higher Order Logics, TPHOLs 2003, volume 2758 of Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.

[23] Derek C. Oppen. Elementary bounds for presburger arithmetic. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 34–37. ACM, 1973.

[24] Derek C. Oppen. A 222pn upper bound on the complexity of Presburger Arithmetic. *Journal of Computer and System Sciences*, 1978.

[25] Igor Ostrovsky. Parallel Programming in .NET 4: Coding Guidelines, 2011.

[26] Harald Prokop. Cache-Oblivious Algorithms, 1999.

[27] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 1991.

[28] C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. ACM, 1978.

[29] Microsoft Research. Practical Parallel and Concurrent Programming: Course Overview. `http://ppcp.codeplex.com/`, 2011.

[30] Robert E. Shostak. On the SUP-INF Method for Proving Presburger Formulas. *J. ACM*, October 1977.

[31] Ryan Stansifer. Presburgerś Article on Integer Airthmetic: Remarks and Translation. Technical report, Cornell University, Computer Science Department, September 1984.

[32] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30, 2005.

Appendix A

Examples of multicore parallelism in F#

A.1 Source code of π calculation

```
module PI

open System
open System.Linq
open System.Threading.Tasks
open System.Collections.Concurrent

let NUM_STEPS = 100000000
let steps = 1.0 / (float NUM_STEPS)

let compute1() =
    let rec computeUtil(i, acc) =
        if i = 0 then acc * steps
        else
            let x = (float i + 0.5) * steps
            computeUtil (i-1, acc + 4.0 / (1.0 + x * x))
    computeUtil(NUM_STEPS, 0.0)

let compute2() =
    let sum = ref 0.0
```

```
    for i in 1..NUM_STEPS do
        let x = (float i + 0.5) * steps
        sum := !sum + 4.0 / (1.0 + x * x)
    !sum * steps

module Parallel =
    let compute1() =
        let sum = ref 0.0
        let monitor = new Object()
        Parallel.For(
            0, NUM_STEPS, new ParallelOptions(),
            (fun () -> 0.0),
            (fun i loopState (local:float) ->
                let x = (float i + 0.5) * steps
                local + 4.0 / (1.0 + x * x)
                ),
            (fun local -> lock (monitor) (fun () -> sum := !sum + local))) |>
             ignore
        !sum * steps

    // Overall best function
    let compute2() =
        let rangeSize = NUM_STEPS / (Environment.ProcessorCount * 10)
        let partitions = Partitioner.Create(0, NUM_STEPS, if rangeSize >= 1
             then rangeSize else 1)
        let sum = ref 0.0
        let monitor = new Object()
        Parallel.ForEach(
            partitions, new ParallelOptions(),
            (fun () -> 0.0),
            (fun (min, max) loopState l ->
                let local = ref 0.0
                for i in min .. max - 1 do
                    let x = (float i + 0.5) * steps
                    local := !local + 4.0 / (1.0 + x * x)
                l + !local),
            (fun local -> lock (monitor) (fun () -> sum := !sum + local))) |>
             ignore
        !sum * steps

let sqr x = x * x

module Linq =
    // LINQ
    let compute1() =
        (Enumerable
            .Range(0, NUM_STEPS)
```

```
        .Select(fun i -> 4.0 / (1.0 + sqr ((float i + 0.5) * steps)))
        .Sum()) * steps

// PLINQ
let compute2() =
    (ParallelEnumerable
        .Range(0, NUM_STEPS)
        .Select(fun i -> 4.0 / (1.0 + sqr ((float i + 0.5) * steps)))
        .Sum()) * steps
```

A.2 Source code of MergeSort

```
module MergeSort

open System
open System.IO
open System.Threading.Tasks
open Microsoft.FSharp.Collections

let split fs =
    let len = Array.length fs
    fs.[0..(len/2)-1], fs.[len/2..]

let rec binarySearch (ls: _ [], v, i, j) =
    if i = j then j
    else
        let mid = (i+j)/2
        if ls.[mid] < v then binarySearch(ls, v, mid+1, j)
        else binarySearch(ls, v, i, mid)

let rec merge (l1: _ [], l2) =
    if l1 = [||] then l2
    elif l2 = [||] then l1
    elif Array.length l1 < Array.length l2 then merge (l2, l1)
    elif Array.length l1 = 1 then
        if l1.[0] <= l2.[0] then Array.append l1 l2
        else Array.append l2 l1
    else
        let head1, tail1 = split l1
        let midVal1 = head1.[Array.length head1 - 1]
        let idx = binarySearch (l2, midVal1, 0, l2.Length)
        let m1, m2 = merge (head1, l2.[..idx-1]), merge (tail1, l2.[idx..])
        Array.append m1 m2

let rec pmergeUtil (l1: _ [], l2, depth) =
    if l1 = [||] then l2
    elif l2 = [||] then l1
    elif Array.length l1 < Array.length l2 then pmergeUtil (l2, l1, depth)
    elif Array.length l1 = 1 then
        if l1.[0] <= l2.[0] then Array.append l1 l2
        else Array.append l2 l1
    elif depth = 0 then merge (l1, l2)
    else
        let head1, tail1 = split l1
        let midVal1 = head1.[Array.length head1 - 1]
        let idx = binarySearch (l2, midVal1, 0, l2.Length)
```

```
        let m1 = Task.Factory.StartNew( fun () -> pmergeUtil (head1, l2.[..idx
            -1], depth-1))
        let m2 = pmergeUtil (tail1, l2.[idx..], depth-1)
        Array.append m1.Result m2

let rec msort ls =
    if Array.length ls <= 1 then ls
    else
        let ls1, ls2 = split ls
        let ls1', ls2' = msort ls1, msort ls2
        merge (ls1', ls2')

let rec pmsortUtil (ls, depth) =
    if Array.length ls <= 1 then ls
    elif depth = 0 then msort ls
    else
        let ls1, ls2 = split ls
        let ls1' = Task.Factory.StartNew(fun () -> pmsortUtil (ls1, depth-1))
        let ls2' = pmsortUtil (ls2, depth-1)
        pmergeUtil (ls1'.Result, ls2', depth)

let rec pmsort ls =
    pmsortUtil(ls, 4)
```

Appendix B

Source code of experiments

B.1 Utilities.fs

```
module Utilities

open System
open System.Threading.Tasks
open System.Collections.Concurrent

type System.Threading.Tasks.Task with
    static member WaitAll(ts) =
        Task.WaitAll [| for t in ts -> t :> Task |]

let ( %| ) a b = (b % a = 0)

let rec gcd(l1, l2) =
    if l2 = 0 then l1 else gcd(l2, l1 % l2)

// Calculate gcd of a list of postive integer.
// Rewrite to use tail recursion.
let gcds ls =
    let rec recGcds ls res =
        match ls with
        | [] -> res
```

```
        | l::ls' -> recGcds ls' (gcd(res, abs(l)))
    match ls with
    | [] -> 1
    | [l]-> abs(l)
    | l::ls' -> recGcds ls' l

let gcda ls =
    Array.fold(fun acc l -> gcd(abs(l), acc)) 1 ls

let lcm(l1, l2) =
    (l1 / gcd(l1, l2)) * l2

// Calculate lcm of a list of positive integer.
// Rewrite to use tail recursion.
let lcms ls =
    let rec recLcms ls res =
        match ls with
        | [] -> res
        | l::ls' -> recLcms ls' (lcm(res, abs(l)))
    recLcms ls 1

let lcma ls =
    Array.fold(fun acc l -> lcm(abs(l), acc)) 1 ls
```

B.2 Term.fs

```
module Term

open Microsoft.FSharp.Collections
open Utilities

type Term = struct
    val constant: int
    val coeffs: int [] // Order of fields matters
    val vars: string []
    new(c, vs, cs) = {constant = c; vars = vs; coeffs = cs}
    override t.ToString() = "(" + string t.constant + ", [" + Array.fold2 (fun
        acc v c -> acc + "(" + string v + "," + string c + ")" + ", " ) "" t.
        vars t.coeffs + "])"
end

let newTerm (c, xs, cs) = Term(c, xs, cs)
let term(c, x) = Term(0, [|x|], [|c|])
let var x = term(1, x)

let constTerm i = Term(i, [||], [||])
let Zero = constTerm 0
let One = constTerm 1
let MinusOne = constTerm -1

let rec merge(n1, n2, l1, l2, t1: Term, t2: Term, acc1: ResizeArray<_>, acc2:
     ResizeArray<_>) =
    if n1 = 0 then
        for i = l2-n2 to l2-1 do
            acc1.Add(t2.vars.[i])
            acc2.Add(t2.coeffs.[i])
    elif n2 = 0 then
        for i = l1-n1 to l1-1 do
            acc1.Add(t1.vars.[i])
            acc2.Add(t1.coeffs.[i])
    else
        let v1 = t1.vars.[l1-n1]
        let c1 = t1.coeffs.[l1-n1]
        let v2 = t2.vars.[l2-n2]
        let c2 = t2.coeffs.[l2-n2]
        if v1 = v2 then
            if c1 + c2 <> 0 then
                acc1.Add(v1)
                acc2.Add(c1+c2)
            merge(n1-1, n2-1, l1, l2, t1, t2, acc1, acc2)
        elif v1 < v2 then
```

```
            if c1 <> 0 then
                acc1.Add(v1)
                acc2.Add(c1)
            merge(n1-1, n2, l1, l2, t1, t2, acc1, acc2)
        else
            if c2 <> 0 then
                acc1.Add(v2)
                acc2.Add(c2)
            merge(n1, n2-1, l1, l2, t1, t2, acc1, acc2)

let (++) (t1: Term) (t2:Term) =
    let l1 = t1.vars.Length
    let l2 = t2.vars.Length
    let acc1 = new ResizeArray<_>(l1 + l2)
    let acc2 = new ResizeArray<_>(l1 + l2)
    merge(l1, l2, l1, l2, t1, t2, acc1, acc2)
    Term(t1.constant + t2.constant, acc1.ToArray(), acc2.ToArray())

let (~~) (t: Term) =
    Term(-t.constant, t.vars, Array.map (fun c -> -c) t.coeffs)

let (--) t1 t2 = t1 ++ ( ~~ t2)

let ( ** ) a (t: Term) =
    if a = 0 then Zero
    elif a = 1 then t
    else
        Term(a*t.constant, t.vars, Array.map (fun c -> a*c) t.coeffs)

let ( */* ) (t: Term) a =
    if a = 0 then invalidArg "Term" "Division by zero"
    elif a = 1 then t
    else
        Term(t.constant/a, t.vars, Array.map (fun c -> c/a) t.coeffs)

let rec findIndex f xs i =
    if i >= Array.length xs then -1
    elif f xs.[i] then i
    else findIndex f xs (i+1)

// Find coefficient of variable x in term t
let findCoeff (t: Term, x) =
    try
        let i = Array.findIndex (fun v -> v = x) t.vars
        t.coeffs.[i]
    with
        | :? System.Collections.Generic.KeyNotFoundException -> 0
```

```
// Filter out variable x in term t
let filterOut (t: Term, x): Term =
    Term(t.constant, t.vars, Array.map2(fun v c -> if v=x then 0 else c) t.vars t.
        coeffs) // Not removing for better performance.

// Substitute variable c*x in term t by a new term tx
let substitute (c, x, tx, t): Term =
    let c0 = findCoeff (t, x)
    if c0 = 0 then t
    else
        let lcm = lcm(abs(c0), abs(c))
        let d = lcm/abs(c0)
        let t' = filterOut (d**t, x)
        t' ++ (d*c0/abs(c)) ** tx

let rec subst (t: Term, xts) =
    match xts with
    | [] -> t
    | (x, tx)::xts' -> subst (substitute (1, x, constTerm tx, t), xts')

let getCoeffsWithoutConst (t: Term) =
    List.ofArray t.coeffs

let getCoeffs (t: Term) =
    t.constant::getCoeffsWithoutConst(t)

// Note: may change for optimization
let redCoeffs (t: Term) =
    let vars = [| for i=0 to t.vars.Length-1 do
                    if t.coeffs.[i] <> 0 then yield t.vars.[i]
                |]
    let coeffs = Array.filter(fun c -> c <> 0) t.coeffs
    let t' = Term(t.constant, vars, coeffs)
    let g = getCoeffs t' |> gcds
    if g = 0 || g = 1 then t' else t' */* g

let isConstTerm (t: Term) =
    Array.isEmpty t.vars || Array.forall (fun c -> c = 0) t.coeffs

let getConst (t: Term) = t.constant
```

B.3 Formula.fs

```
module Formula

open Microsoft.FSharp.Collections
open Utilities
open Term

type CompType =
    | EQ
    | UEQ
    | GT

type Formula =
    | TT
    | FF
    | C of Term * CompType
    | D of int * Term
    | Not of Formula
    | And of Formula list
    | Or of Formula list
    | SAnd of Formula * (string * int) list
    | SOr of Formula * (string * int) list (* Avoid expanding formula by
        introducing variable and its range of value *)
    | E of string list * Formula
    | A of string list * Formula
```

B.4 PAGenerator.fs (excerpt)

```
// Generate PA Fragments for testing Cooper algorithm
module PAGenerator

// Formulate pigeon hole principle in propositional logic (N pigeons and K holes)

let rec tab n f =
    match n with
    | x when x <= 0 -> []
    | _ -> f n:: tab (n-1) f

// Pigeon i is in hole k
let X i k = A( L ("x(" + string i + "," + string k + ")"), true)
let Y i k = A( C(Zero, var ("x" + string i + "_" + string k + ""), LE), true)
let Z i k = A( C(Zero, var ("x" + string i + "_" + string k + ""), EQ), true)
let T i k = A( D(2, var ("x" + string i + "_" + string k + "")), true)

// If pigeon i is in hole k so no one else is in hole k
let F pred (i, k) (N, K) = (pred i k) => (And (tab N (fun i' -> if i = i' then TT
     else Not (pred i' k))))

// Apply F for all i and k
let Fall pred (N, K) = And (tab N (fun i -> And (tab K (fun k -> F pred (i, k) (N,
     K)))))

// Pigeon i has assigned a hole
let G pred i (N, K) = Or (tab K (fun k -> pred i k))

// All pigeons have been assigned holes
let Gall pred (N, K) = And (tab N (fun i -> G pred i (N, K)))

// A pigeon is only in one hole
let H pred (i, k) (N, K) = (pred i k) => (And (tab K (fun k' -> if k = k' then TT
     else Not (pred i k'))))

// Every pigeon has exactly one hole
let Hall pred (N, K) = And (tab N (fun i -> And (tab K (fun k -> H pred (i, k) (N,
     K)))))

let pigeon pred (N, K) = And [Fall pred (N, K) ; Gall pred (N, K) ; Hall pred (N, K)]

let pigeonY = pigeon Y
let pigeonZ = pigeon Z
let pigeonT = pigeon T

let generateFormula pigeonPred (N, K, Q) =
```

```
    let rec takeLast(ls, q) =
        if List.length ls <= q then ls
        else takeLast(List.tail ls, q)

    let qfForm = (N, K) |> pigeonPred |> gnfToFormula
    let quantifiers = List.fold (fun acc (i, k) ->
                                        ("x" + string i + "_" + string k)::acc )
                                        [] (List.fold (fun acc i ->
                                                            acc @ (List.map (fun k ->
                                                                  (i, k)) (tab K (fun k
                                                                  -> k))))
                                                      [] (tab N (fun i -> i)))
    if Q = 0 then qfForm
    else
        E(takeLast(quantifiers, Q), qfForm)

let generatePigeonYFormula = generateFormula pigeonY
let generatePigeonZFormula = generateFormula pigeonZ
let generatePigeonTFormula = generateFormula pigeonT
```

B.5 Cooper.fs (excerpt)

```
// Quantifier elimination by means of Cooper algorithm
module Cooper

open System
open System.Collections
open System.Threading.Tasks
open System.Collections.Concurrent
open Microsoft.FSharp.Collections
open Utilities
open Term
open Formula

// Input: quantifier-free formulas
// Output: negation is pushed into literals.
let rec nnf formula =
    match formula with
    | C(_, _)
    | D(_, _)
    | TT
    | FF -> formula
    | And fs -> And (List.map nnf fs)
    | Or fs -> Or (List.map nnf fs)
    | SAnd(f, vr) -> SAnd (nnf f, vr)
    | SOr(f, vr) -> SOr (nnf f, vr)
    | Not f -> nnnf f
    | _ -> invalidArg "nnf" "Unwellformed"

and nnnf formula =
    match formula with
    | TT -> FF
    | FF -> TT
    | C(t, ct) -> match ct with // Preserve wellformness of formulas
                  | EQ -> C(t, UEQ)
                  | UEQ -> C(t, EQ)
                  | GT -> C(One -- t, GT)
                  | _ -> invalidArg "nnnf" "Comparison"
    | D(i, t) -> Not formula
    | And fs -> Or (List.map nnnf fs)
    | Or fs -> And (List.map nnnf fs)
    | SAnd(f', vr) -> SOr (nnnf f', vr)
    | SOr(f', vr) -> SAnd (nnnf f', vr)
    | Not f' -> nnf f'
    | _ -> invalidArg "nnnf" "Unwellformed"
```

```
let rec mapUntil(func, condVal: Formula, combFunc, fs: Formula list, acc: _ list)
    =
    match fs with
    | [] -> combFunc acc
    | f::fs'-> let f' = func f
               if f' = condVal then condVal else mapUntil(func, condVal,
                   combFunc, fs', f'::acc)

let rec genInfFormula (x, leftProjection, lcm) form =
    match form with
    | C(t, ct) -> match ct with
                  | EQ -> if findCoeff (t, x) <> 0 then FF
                          else
                              if lcm = 1 then C(filterOut(t, x), ct) else
                                  form
                  | UEQ -> if findCoeff (t, x) <> 0 then TT
                          else
                              if lcm = 1 then C(filterOut(t, x), ct) else
                                  form
                  | GT -> let c = findCoeff (t, x)
                          if c > 0 && leftProjection then FF
                          elif c < 0 && not leftProjection then FF
                          elif c <> 0 then TT
                          else
                              if lcm = 1 then C(filterOut(t, x), ct) else
                                  form
                  | _ -> if lcm = 1 then C(filterOut(t, x), ct) else form
    | D(i, t) -> if lcm = 1 then D(i, filterOut(t, x)) else form
    | And fs -> mapUntil(genInfFormula (x, leftProjection, lcm), FF, And, fs, [])
    | Or fs -> mapUntil(genInfFormula (x, leftProjection, lcm), TT, Or, fs, [])
    | Not f -> match genInfFormula (x, leftProjection, lcm) f with
                  | TT -> FF
                  | FF -> TT
                  | f' -> Not f'
    | SAnd(f, vr) -> match genInfFormula (x, leftProjection, lcm) f with
                  | TT -> TT
                  | FF -> FF
                  | SAnd(f', vr') -> SAnd(f', vr'@vr)
                  | f' -> SAnd(f', vr)
    | SOr(f, vr) -> match genInfFormula (x, leftProjection, lcm) f with
                  | TT -> TT
                  | FF -> FF
                  | SOr(f', vr') -> SOr(f', vr'@vr)
                  | f' -> SOr(f', vr)
    | _ -> form

let rec substituteFormula (c, x, tx) formula =
```

```
    match formula with
    | C(t, ct) -> C(substitute (c, x, tx, t), ct)
    | D(i, t) -> D(i, substitute (c, x, tx, t))
    | And fs -> fs |> List.map (substituteFormula (c, x, tx)) |> And
    | Or fs -> fs |> List.map (substituteFormula (c, x, tx)) |> Or
    | SAnd(f, vr) -> (substituteFormula (c, x, tx) f, vr) |> SAnd
    | SOr(f, vr) -> (substituteFormula (c, x, tx) f, vr) |> SOr
    | Not f -> substituteFormula (c, x, tx) f |> Not
    | _ -> formula

let rec genTermFormula (x, leftProjection, aLits, bLits, lcm, form) =
    let newX = if lcm = 1 then One else var x
    if leftProjection then
        bLits |> List.map (fun (c, b) -> match substituteFormula (c, x, b ++
            newX) form with
                                          | And fs -> if c = 1 then And fs else
                                               And ((D(c, b ++ newX))::fs)
                                          | f -> if c = 1 then f else And [f; D(c,
                                               b ++ newX)]
                                          )
    else
        aLits |> List.map (fun (c, a) -> match substituteFormula (c, x, a --
            newX) form with
                                          | And fs -> if c = 1 then And fs else
                                               And ((D(c, a -- newX))::fs)
                                          | f -> if c = 1 then f else And [f; D(c,
                                               a -- newX)]
                                          )

let elimVariable x formula =
    // Choice of left projection or right projection depends on the number of literals
        .
    let divCoeffs, aLits, bLits = retrieveInfo (x, formula)

    let leftProjection = aLits.Length >= bLits.Length
    //let _ = if leftProjection then bLits.Length |> printfn "Left projection:%i" else
        aLits.Length |> printfn "Right projection:%i"
    let lcm = divCoeffs |> lcms

    match formula with
    | SOr(f', vr) -> match genInfFormula (x, leftProjection, lcm) f',
        genTermFormula (x, leftProjection, aLits, bLits, lcm, f') with
                         | TT, _ -> TT
                         | FF, []-> FF
                         | FF, fs-> if lcm = 1 then (fs |> Or, vr) |> SOr else (fs
                             |> Or, (x, lcm)::vr) |> SOr
```

```
                    | f, [] -> if lcm = 1 then (f, vr) |> SOr else (f, (x, lcm)::vr
                        ) |> SOr
                    | f, fs -> if lcm = 1 then (f::fs |> Or, vr) |> SOr else (f::
                        fs |> Or, (x, lcm)::vr) |> SOr
    | _ -> match genInfFormula (x, leftProjection, lcm) formula,
        genTermFormula (x, leftProjection, aLits, bLits, lcm, formula) with
                    | TT, _ -> TT
                    | FF, []-> FF
                    | FF, fs-> if lcm = 1 then fs |> Or else (fs |> Or, [(x, lcm
                        )]) |> SOr
                    | f, [] -> if lcm = 1 then f else (f, [(x, lcm)]) |> SOr
                    | f, fs -> if lcm = 1 then f::fs |> Or else (f::fs |> Or, [(x,
                        lcm)]) |> SOr

// The formula is quantifier-free.
let rec elimQuantifier x formula =
    match formula with
    | SAnd(f, vr) -> (f |> elimQuantifier x, vr) |> SAnd
    | SOr(f, vr) -> (f |> elimQuantifier x, vr) |> SOr
    | Or fs -> fs |> List.map (elimQuantifier x) |> Or
    | _ -> formula |> reduce |> elimVariable x

// Eliminate all quantifiers, one by one.
let rec cooper formula =
    match formula with
    | Not f -> f |> cooper |> Not
    | And fs -> fs |> List.map cooper |> And
    | Or fs -> fs |> List.map cooper |> Or
    | SAnd(f, vr) -> (f |> cooper, vr) |> SAnd
    | SOr(f, vr) -> (f |> cooper, vr) |> SOr
    | E(xs, SOr(f, vr))
                -> SOr(cooper (E(xs, f)), vr)
    | E(xs, Or fs) -> fs |> List.map (fun f -> cooper (E(xs, f))) |> Or
    | E(xs, f) -> List.fold (fun acc x -> (elimQuantifier x acc)) (nnf (cooper
        f)) xs
    | A(xs, SAnd(f, vr))
                -> SAnd(cooper (A(xs, f)), vr)
    | A(xs, And fs) -> fs |> List.map (fun f -> cooper (A(xs, f))) |> And
    | A(xs, f) -> Not (List.fold (fun acc x -> (elimQuantifier x acc)) (nnf (
        cooper (Not f))) xs)
    | _ -> formula

let parMap func fs =
    let fs' = List.map (fun f -> Task.Factory.StartNew(fun() -> func f)) fs
    Task.WaitAll(fs')
    List.map (fun (t': Task<_>) -> t'.Result) fs'
```

```
let rec elimQuantifierParallel x formula =
    //printfn "var0=%s" x
    match formula with
    | SAnd(f, vr) -> (f |> elimQuantifierParallel x, vr) |> SAnd
    | SOr(f, vr) -> (f |> elimQuantifierParallel x, vr) |> SOr
    | Or fs -> //printfn "elim.var=%s, fs=%i" x fs.Length
                    fs |> parMap (elimQuantifier x) |> Or
    | _ -> formula |> reduce |> elimVariable x

// Eliminate all quantifiers, one by one.
let rec cooperParallel formula =
    match formula with
    | Not f -> f |> cooperParallel |> Not
    | And fs -> fs |> List.map cooperParallel |> And
    | Or fs -> fs |> List.map cooperParallel |> Or
    | SAnd(f, vr) -> (f |> cooperParallel, vr) |> SAnd
    | SOr(f, vr) -> (f |> cooperParallel, vr) |> SOr
    | E(xs, SOr(f, vr))
                -> SOr(cooperParallel (E(xs, f)), vr)
    | E(xs, Or fs) -> //printfn "EOr.xs=%i, fs=%i" xs.Length fs.Length
                    fs |> parMap (fun f -> cooperParallel (E(xs, f))) |> Or
    | E(xs, f) -> List.fold (fun acc x -> (elimQuantifierParallel x acc)) (nnf
        (cooperParallel f)) xs
    | A(xs, SAnd(f, vr))
                -> SAnd(cooperParallel (A(xs, f)), vr)
    | A(xs, And fs) -> //printfn "AAnd.xs=%i, fs=%i" xs.Length fs.Length
                    fs |> parMap (fun f -> cooperParallel (A(xs, f))) |> And
    | A(xs, f) -> Not (List.fold (fun acc x -> (elimQuantifierParallel x acc))
        (nnf (cooperParallel (Not f))) xs)
    | _ -> formula

let elimQuantifiers = cooper >> reduce
let elimQuantifiersParallel = cooperParallel >> reduce

//
// Evaluation part
//
let cartesian lss =
    let k l ls = [ for x in l do
                    for xs in ls -> x::xs ]
    List.foldBack k lss [[]]

let genRangeArray vr =
    vr |> List.map (fun (v, r) -> List.init r (fun i -> (v, i))) |> cartesian |>
        List.toArray

// Substitute a list of variables and associated values to a formula
```

```
// Suppose the formula is quantifier-free
let rec evalFormula xts formula =
    match formula with
    | C(t, ct) -> match subst (t, xts) with
                  | t' -> if isConstTerm t' then
                              let c = getConst t'
                              match ct, c with
                              | EQ, 0 -> TT
                              | UEQ, x when x <> 0 -> TT
                              | GT, x when x > 0 -> TT
                              | _, _ -> FF
                          else invalidArg "evalFormula" (string xts)

    | D(i, t) -> match subst (t, xts) with
                 | t' -> if isConstTerm t' then
                             let c = getConst t'
                             if i %| c then TT else FF
                         else invalidArg "evalFormula" (string xts)

    | And fs -> if List.exists (fun f -> evalFormula xts f = FF) fs then FF else
         TT
    | Or fs -> if List.exists (fun f -> evalFormula xts f = TT) fs then TT else
         FF
    | Not f -> match evalFormula xts f with
               | TT -> FF
               | FF -> TT
               | f' -> invalidArg "evalFormula" (string xts)
    | TT -> TT
    | FF -> FF
    | _ -> invalidArg "evalFormula" (string xts)

/// Partition functions
let partitionEval groundVal rangeArray formula =
    let len = Array.length rangeArray
    let loopResult = Parallel.For(0, len, fun i (loopState: ParallelLoopState)
        ->
                                        if evalFormula rangeArray.[i] formula =
                                            groundVal then
                                                    loopState.Stop()
                                                else
                                                    ()
                                        )
    not (loopResult.IsCompleted || loopResult.LowestBreakIteration.HasValue)

// Break the array to balance chunks
let partitionBalanceEval groundVal rangeArray formula =
    let len = Array.length rangeArray
```

```
    let source = [|0..len−1|]
    let partitions = Partitioner.Create(source, true)
    let loopResult = Parallel.ForEach(partitions, fun i (loopState:
         ParallelLoopState) −>
                                                        if evalFormula
                                                             rangeArray.[i]
                                                             formula =
                                                             groundVal then
                                                            loopState.Stop()
                                                        else
                                                            ()
                                    )
    not (loopResult.IsCompleted || loopResult.LowestBreakIteration.HasValue)

// Sequential
let seqEval groundVal rangeArray formula = rangeArray |> Array.exists(fun r
     −> evalFormula r formula = groundVal)

let evalSAnd vr formula =
    if vr = [] then formula
    else
        if seqEval FF (genRangeArray vr) formula then FF else TT

let evalSOr vr formula =
    if vr = [] then formula
    elif seqEval TT (genRangeArray vr) formula then TT else FF

let eval formula =
    let rec evalUtil vr0 formula =
        match formula with
        | Not f −> f |> evalUtil vr0 |> Not
        | And fs −> fs |> List.map (evalUtil vr0) |> And
        | Or fs −> fs |> List.map (evalUtil vr0) |> Or
        | SAnd(f, vr) −> evalSAnd (vr@vr0) (evalUtil (vr@vr0) f)
        | SOr(f, vr) −> evalSOr (vr@vr0) (evalUtil (vr@vr0) f)
        | _ −> formula
    evalUtil [] formula


// Parallel
let pevalSAnd vr formula =
    if vr = [] then formula
    elif partitionBalanceEval FF (genRangeArray vr) formula then FF else TT

let pevalSOr vr formula =
    if vr = [] then formula
    elif partitionBalanceEval TT (genRangeArray vr) formula then TT else FF
```

```
let peval formula =
    let rec pevalUtil vr0 formula =
        match formula with
        | Not f -> f |> pevalUtil vr0 |> Not
        | And fs -> fs |> List.map (pevalUtil vr0) |> And
        | Or fs -> fs |> List.map (pevalUtil vr0) |> Or
        | SAnd(f, vr) -> pevalSAnd (vr@vr0) (pevalUtil (vr@vr0) f)
        | SOr(f, vr) -> pevalSOr (vr@vr0) (pevalUtil (vr@vr0) f)
        | _ -> formula
    pevalUtil [] formula
```

B.6 OmegaTest.fs

```
ï»¿module OmegaTest

open Microsoft.FSharp.Collections
open Utilities
open Term

// Input: a list of conjunctions and a list of disjunction pairs.
// Output: do cartesian products on the input.
let cartesian (cons, diss) =
    let rec cartesianUtil = function
      | [] -> [cons]
      | L::Ls -> cartesianUtil Ls |> List.collect (fun C -> L |> List.map (fun
          x -> x::C))
    cartesianUtil diss

exception OmegaTestFail

type CoeffTerm = struct
    val coeff: int
    val term: Term
    new (c, t) = {coeff = c; term = t}
end

// Product all pairs of opposite literals
let merge(outs, lowers: CoeffTerm list, uppers: CoeffTerm list) =
    let ins = [for l in lowers do
                    for u in uppers do
                        if l.coeff = 1 || u.coeff = 1 then
                            yield ((l.coeff ** u.term ++ u.coeff ** l.term) --
                                One)
                        else raise OmegaTestFail]
    ins@outs

let project(x, fs) =
    let outs = List.filter (fun t -> findCoeff(t, x) = 0) fs

    let lowers = [
                  for t in fs do
                    let c = findCoeff(t, x)
                    if c > 0 then yield CoeffTerm(c, t)
                    ]
    let uppers = [
                  for t in fs do
                    let c = findCoeff(t, x)
                    if c < 0 then yield CoeffTerm(-c, t)
```

```
                    ]

    if lowers = [] || uppers = [] then outs
    else
        merge(outs, lowers, uppers)

let omegaTest(xs, fs) =
    let rec omegaTestUtil(xs, fs) =
        match xs with
        | [] -> xs, fs
        | x::xs'-> try
                        omegaTestUtil(xs', project(x, fs))
                    with
                        OmegaTestFail -> xs, fs
    omegaTestUtil(xs, fs)

let resolve(xs, tll) =
    printfn "Resolving %i Omega Tests" (List.length tll)
    List.map (fun tl -> omegaTest(xs, tl)) tll

let resolveParallel(xs, tll) =
    printfn "Resolving %i Omega Tests" (List.length tll)
    List.ofSeq (PSeq.map (fun tl -> omegaTest(xs, tl)) tll)
```

B.7 OmegaTestArray.fs

```
ï»¿module OmegaTestArray

open System.Threading.Tasks
open Utilities
open Term

// Input: a list of conjunctions and a list of disjunction pairs.
// Output: do cartesian products on the input.
let cartesian (cons, diss) =
    let k l ls = [| for x in l do
                    for xs in ls -> Array.append [|x|] xs |]
    Array.foldBack k diss [|cons|]

exception OmegaTestFail

type CoeffTerm = struct
    val coeff: int
    val term: Term
    new (c, t) = {coeff = c; term = t}
end

let merge(outs, lowers: CoeffTerm [], uppers: CoeffTerm []) =
    let ins = [|for l in lowers do
                  for u in uppers do
                    if l.coeff = 1 || u.coeff = 1 then
                      yield ((l.coeff ** u.term ++ u.coeff ** l.term) --
                          One)
                    else raise OmegaTestFail|]
    Array.append ins outs

let project(x, fs) =
    let outs = [|
                 for t in fs do
                   let c = findCoeff(t, x)
                   if c = 0 then yield t
                   |]

    let lowers = [|
                 for t in fs do
                   let c = findCoeff(t, x)
                   if c > 0 then yield CoeffTerm(c, t)
                   |]
    let uppers = [|
                 for t in fs do
                   let c = findCoeff(t, x)
```

```
                    if c < 0 then yield CoeffTerm(-c, t)
                    |]

    if lowers = [||] || uppers = [||] then outs
    else
        merge(outs, lowers, uppers)


let omegaTest(xs, fs) =
    let rec omegaTestUtil(xs, fs) =
        match xs with
        | [] -> xs, fs
        | x::xs'-> try
                        omegaTestUtil(xs', project(x, fs))
                    with
                        OmegaTestFail -> xs, fs
    omegaTestUtil(xs, fs)

let n = 16

let resolvePadding(xs, tll: _ []) =
    //printfn "Resolving %i Omega Tests" (Array.length tll)
    let arr = Array.init tll.Length (fun _ -> [], [||])
    for i=0 to tll.Length/n-1 do
        arr.[n*i] <- omegaTest(xs, tll.[n*i])
    arr

let resolveParallelPadding(xs, tll: _ []) =
    //printfn "Resolving %i Omega Tests" (Array.length tll)
    let arr = Array.init tll.Length (fun _ -> [], [||])
    Parallel.For(0, tll.Length/n,
        fun i -> arr.[n*i] <- omegaTest(xs, tll.[n*i]))|> ignore
    arr
```