Compiler Paralization Design and Implementation using OpenMP

Tongjai Yampaka

Report in Digital Systems: 1st semester 2016

***Abstract*—Compilation is a process that translates a source code in one language (the source language) into another language (the object or target language). The specification design in language is important for building the compiler. Recently, many applications tried to find methods to improve the execution time. The compiler parallelization design is introduce. In this report shows two designs (the specification design and the parallel implementation design). The result shows that the compiler design approaches to compile the case study source code to the target code (S-code) and some compiler phases can be done in parallel execution.**

Keywords; formatting; compiler design; parallel compiler; parallel programming

I. INTRODUCTION (HEADING 1)

The computer system is made of hardware and software [1]. The hardware understands a language, which humans cannot understand. The programs in high-level language, which is easier for the programmer to understand. Language Processing System are a series of tools and OS components to get the code that can be used by the machine. A compiler is a program that converts high-level language to intermediate language such as assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. The procedure to build the compiler starts with the specification design such as the regular expression design, and context-free grammar.

Recently, multicore processors have been proposed as a solution to provide both performance and scalability. Many application tried to use this advantage. This report introduces the compiler parallelization design in case study instruction. This work divides in two design. The first, the specification of compiler language design. The second, the parallel compiler implementation design.

This paper is organized as follows. Section 2 introduces general background on compiler design. Section 3 the background on basic parallelization techniques in openMP. Section 4 shows the methodology. Section 5 shows the experimental result and Section 6 conclusion.

II. COMPILER DESIGN

A. Bsic compiler design.

Nowadays, the program is written in a high-level programming language [2] such as JAVA, C, or Python. The compiler translates the high-level to the low-level language what is called machine language. In another mean, compilation is a process that translates a program in one language (the source language) into another language (the object or target language) [3].

B. The phases of compiler.

The work of compiler split into several phases. These phases operate in sequence.

- Lexical analysis: This is the initial part of reading and analyzing the program text. The text is read and split into tokens.
- Syntax analysis: This phase takes the list of tokens produced by the lexical analysis and assign these in a syntax tree. This phase is often called parsing.
- Semantic Analysis: This phase takes its input from the syntax analysis phase (the form of a parse tree) and determines if the input has a well-defined meaning.
- Intermediate code generation: The program is translated to a simple machine-independent intermediate language.
- Register allocation: The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.
- Machine code generation: The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.
- Assembly and linking: The assembly-language code is translated into binary representation and addresses of variables, functions are determined.

The first three phases are called the frontend of the compiler and the last three phases are called the backend. The intermediate code generation are called the middle. In this report shows the interpreter design (the frontend and the middle design).

III. PARALLEL PROGRMMING CONCEPT

Parallelization is another optimization technique. The goal is to reduce the execution time. Parallel computing uses the multiple compute resources to solve a computational problem. A problem is broken into discrete parts that can be solved concurrently. Each part is further broken down into a series of instructions. Instructions from each part execute on different processors [4].

A. Parallel programming with OpenMP API.

The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel [4]. The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. In C/C++, OpenMP directives are specified by using the #pragma mechanism provided by the C and C++ standards. The syntax of an OpenMP directive is as follows:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
        structured-block
```

The directives allow the user to mark areas of the code, such as do, while or for loops, which are suitable for parallel processing. OpenMP allows the program to request any number of threads of execution. If your system has four processors available, maybe 4 threads is what you want.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */

}
```

Figure 1. Example of C parallel code in OpenMP

Figure 1 the code print “Hello World” from the number of thread. The execution can be done not using loop. In this report, the parallel threads are used in parallel compiler.

The parallel run time is defined as the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution.

Notation: Serial run time T_s, parallel run time T_p

Speedup = T_s / T_p

The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on *p* processors.

IV. METHODOLOGY

The proposed method comprised of (1) the design of compiler (2) build the parallel compiler using OpenMP (3) report the experimental results.

A. The design of compiler.

1) Lexical analysis: The word “lexical” in the compiler means “split the words”[3]. The lexical analyzer or scanner is the sequences of characters into lexemes, and outputs (to the syntax analyser) a sequence of tokens. Here:

TABLE I. LEXICAL PATTERNS

Lexeme	Token	Token no.	Pattern
def	def	10	d, e, f
ifelse	ifelse	20	i, f, e, l, s, e
main	main	12	m, a, i, n
print	print	0	p, r, i, n, t
∗ | – | + | ==	oper	32, 31, 30, 11	∗ or – or + or ==
a-z A-Z	identifier	40	a-z A-Z
0…9	digit	50	0…9

In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens. For lexical analysis or scanner, specifications are written using regular expressions.

progrm = def fun-name identifier expression
def = \b**def**?\b
fun-name = [a-zA-Z]+
identifier = [a-zA-Z]+
digit = [0-9]+
expression = ∗ | – | + | == | identifier | digit | ifesle

2) Syntax analysis: Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens. It is the sequence of tokens that are passed as output to the syntax analyzer. The notation use for manipulation is context-free grammars.

progrm = def fun-name | identifier | (expression)
  | main (expression)
expression = (ifelse (expression)) | (expression)
  = (∗ | – | + |) expression
  = fun-name (expression)
  = identifier | digit
  = print (expression)
  = (= identifier digit | identifier identifier)
  = (== identifier identifier | digit)
fun-name = identifier
identifier = [a-zA-Z]+
digit = [0-9]+

Parse Trees and Syntax Trees. The structure is a parse tree or a syntax tree. A parse tree is tree that represents the tokens into phrases[3]. A syntax tree the operators appear the interior nodes. The construction of a parse tree is a basic activity in compiler-writing.

For example: def fac n

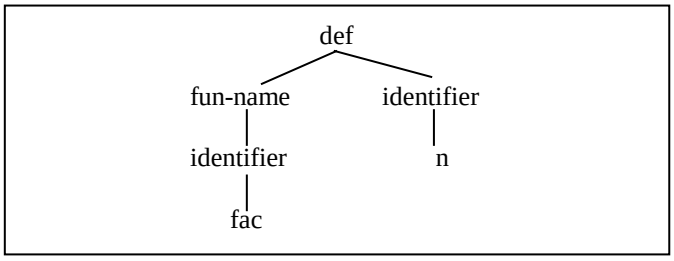

Figure 2. Example of Parse Tree

3) Semantic analysis: A semantic analyser determine the input has a well-defined meaning. Semantic analysers are mainly concerned with type checking and type coercion basedon type rules. For example *(= a b)* is expression based on rule *(= identifier digit | identifier identifier)* that means assignmet type *(a = b)*.

4) Intermediate code generation: The final goal of a compiler is to get programs written in a high-level language to run on a computer[2]. This phase translate directly from the high-level abstract syntax to machine code. Many compilers use a medium-level language between the high-level language and the very low-level machine code. In this report use S-code instruction [5]. This design uses stack in translated table in expression. For example instruction:

def fac n → def fac n
ifelse == n 0 → ifelse == n 0
fac * n - n 1 → 1 n – n * fac
def main → def main
fac n print → print n fac

TABLE II. CODE GENERATION

Target Code	Translator	Intermediate code
def fac	labelled instruction	:label
n	an assignment value	mov r1 r2
ifelse	loop instruction	:loop
== n 0	loop condition	eq r1 r1 0
		jt r1 main
		jf r1 trap 0
		jmp label
1 n –	expression	sub r2 r1 1
n *	expression	mul r1 r1 r2
fac	call label	jmp label
def main	labelled instruction	:main
print	expression	trap 1
n	expression	mv r1 r1 n
fac	call label	jmp label

B. Implementation in thread parallelisms.

In this report uses thread parallelisms in openMP. Only the scanner and code generation were done by thread parallelisms. The target code was read from text file to array before calculation. Each thread calculations can be performed on either the same or different sets of data.

thread-id[i]	data[i]	output token[i]
thread-id [0]	def	10
thread-id [1]	fac	40
thread-id [2]	6	50
.	.	.
.	.	.
thread-id [n]	n	n

```
#pragma omp for ordered schedule(dynamic, 10)
//use 10 threads
for (int n = 0; n < 10; ++n)
  {
//sorting thread id before run parallel
       #pragma omp ordered  tid = n;
       ret[tid] = mylex2(tid, arrayc[tid]);
       mytoken[tid] = ret[tid];

  }
```

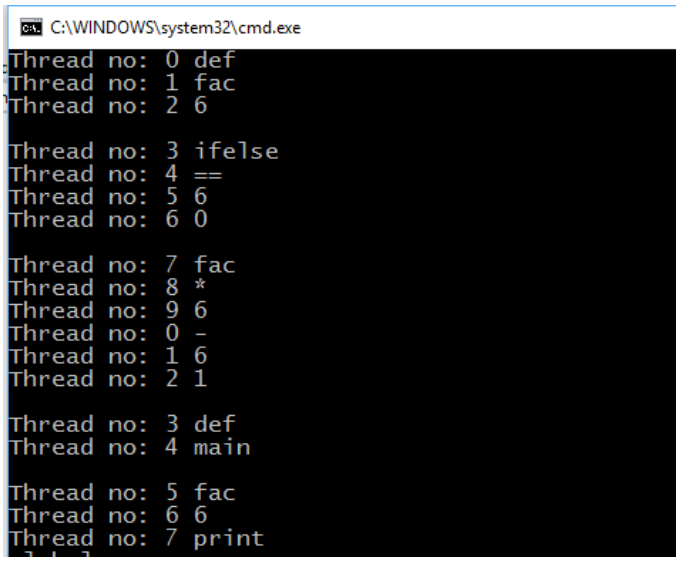

Figure 3. Scanner result

The execution can be done not using loop. The code generation concern about the sequence of token but it is can be done by task parallelisms using dividing one line per thread. Each thread match the source code to the intermediate code.

thread-id[i]	data[i]	output token[i]
thread-id [0]	def fac 6	:label
		mov r1 r2
thread-id [1]	ifelse == n 0	eq r1 r1 0
		jt r1 main
		jf r1 trap 0
		jmp label
.	.	.
.	.	.
thread-id [n]	n	n

Sometimes it is handy to indicate that "this and this can run in parallel". The sections setting is just for that

```
#pragma omp parallel sections
// starts a new team
	{
	#pragma omp section
		{ myparser(mytoken);}
	#pragma omp section
		{gen(mytoken); }
	}
```

The parallel can separate in section. This code indicates that any of the tasks myparser and codegen may run in parallel. Each work is done exactly once.

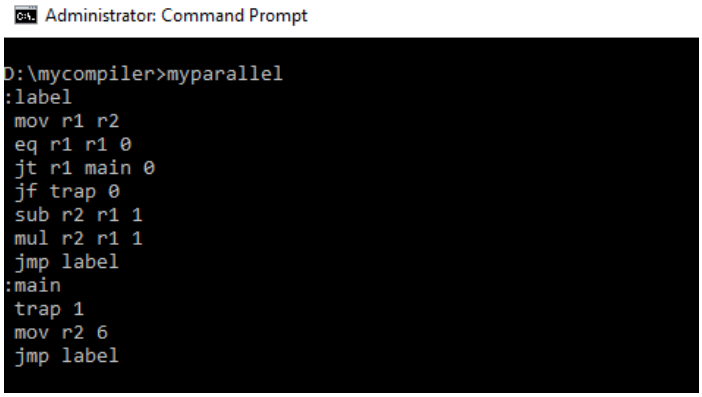

Figure 4. Code generator result

V. EXPERIMENTS.

1) Design: From the little target code in case study, this report designs the compiler in four phases. The lexical analyzer or scanner splits the word into the token using regular expression design. The syntax analyzer assigns these tokens into the syntax tree using grammar rules (context free grammar) and the semantic analyzer determines the input has a well-defined meaning. Then, the intermediat code generation phase directly translates the target code into the intermediate code (S-code).

2) Implementation: In experiment, the compiler processing can be done in parallel. The two phases in scanner and code generation can be done by thread parallelisms in openMP. The syntax analysis concern about the ordering of words so the grammar checked phase must execute in ordering. The parallelisms can discard the loop instruction (for..loop, while…loop) by using the thread parallelisms thus, the time performance is reduced.

VI. CONCLUSION

This report presents compiler penalization design and implementation in openMP. The design based on the case study code in easy and small instruction. Not at all phase in design are parallelism such as syntax analysis which phase concern in ordering correctly instruction. Only two parallel phases in compiler construction are introduced. However, the target code was read from text file to array before parallel execution. Therefore, it may be the case that the data access time will dominate the total execution time if the task and the data are not carefully divided.

REFERENCES

[1] "Tutorialspoint," 2014. [Online]. Available: https://www.tutorialspoint.com/compiler_design/compiler_design_tutorial.pdf. [Accessed 19 10 2016].

[2] C. W. Matt Poole, Compilers Course notes for module CS 218, 2007.

[3] T. Æ. Mogensen, Basics of Compiler Design, 2010.

[4] A. R. Board, "The OpenMP® API specification for parallel programming," November 2015. [Online]. Available: http://www.openmp.org/mp-documents/openmp-4.5.pdf. [Accessed 19 10 2016].

[5] P. Chongstitvatana, "S-code," 2016. [Online]. Available: https://www.cp.eng.chula.ac.th/~piak/project/som/s-code.htm. [Accessed 19 10 2016].