USING EFFICIENT PROGRAMMING TO CREATE EFFICIENT PROGRAMS: COMBINING EASILY PARALLELIZABLE LANGUAGES WITH EFFICIENT C/C++/ASM LIBRARIES

Saša N. Malkov

Faculty of Mathematics, University of Belgrade, Studentski Trg 16, 11000 Belgrade, Serbia
e-mail: smalkov@matf.bg.ac.rs

Abstract:

In contemporary software development, it is often not enough for programs to be correct and efficient. The number of developers is not growing as fast as the need for new programs, so, in order to be as efficient as possible it is important to speed up the development process. One of the steps in that direction is the use of so-called scripting languages. However, although scripting languages provide for more efficient programming, they do not usually result in particularly efficient programs.

We present our work on two techniques for improving the efficiency of programs written in scripting languages: (1) combining the main programs written in scripting languages with some parts of the program written in C/C++/ASM, and (2) techniques for implicit parallelization of programs or some parts of the programs written in scripting languages.

Our research is conducted on the programming language Wafl. The results are already used in bioinformatics research projects at the Faculty of Mathematics.

Keywords: combining programming languages, marshaling, parallelization, implicit parallelization, Wafl

1. Introduction

1.1 Program Efficiency

Program efficiency is, in addition to correctness, one of the most important characteristics of a program. By program efficiency we traditionally mean that programs are able to do work with minimal use of computer system resources – primarily considering CPU time and runtime memory. In order for a program to be efficient, it is crucial that it is based on a good and efficient algorithm, but it is also very important that the programming language and accompanying tools support its efficient execution. Since the advent of multiprocessor computers and especially multi-core CPUs, the program efficiency usually includes the ability to execute some parts of the program concurrently [1].

1.2 Programming Efficiency

In contemporary software development, it is often not enough for programs to be correct and efficient, but the process of program development also has to be as efficient as possible in terms of developers` time. The number of developers is not growing as fast as the need for new programs, so it is important to simplify and shorten the development cycle in order to accelerate development. This is why so-called *scripting languages* [2] are often used today. Scripting programming languages are usually interpreted (rather than compiled) and often have somewhat simpler and less formal syntax. Scripting languages are especially useful for developing programs of smaller size and complexity, and even more so in cases where it is

necessary to write a large number of such programs. Typical examples of such use are data preparation, data processing and analysis of data mining results.

1.3 Programming Language Wafl

The research we present here is conducted on the *Wafl* programming language [3]. Wafl is a strongly typed functional programming language with implicit type inference. It is a general purpose language, although it was originally designed for Web development. It was designed and developed by the author, which facilitates the implementation of various researches.

2. Materials and Methods

The cost of efficient programming in script languages is usually paid by obtaining less efficient programs. In this paper, we present our research on how to enable the writing of efficient programs with efficient programming in script languages.

Two main ways to increase program efficiency are discussed:

- combining scripting languages with efficient languages, by using the parts of the programs written in C / C ++ / assembler in main programs written in Wafl, and
- implicit (or as implicit as possible) parallelization of Wafl programs.

2.1 Combining Wafl Programs and C/C++/Asm Libraries

One of the most direct ways to make a program more efficient is to perform optimization by writing the most sensitive parts of the program in machine language (assembler). When programming in a scripting language, it is often sufficient to write such parts of the program in C/C++. Writing selected parts of the program in C/C++ has practically the same effect as extending the scripting language function library. Such a combination, if well designed, can lead to very high performance efficiency, even virtually indistinguishable from programs written entirely in C/C++.

Combining the parts of programs written in different languages brings with it several problems, the most complex of which is transforming the memory representation of data (arguments and results) from a form understandable in one language to a form understandable in another language [4], so-called marshaling. It can be done in several different ways, including:

- explicit marshaling in program code – explicit writing of program elements that transform data when writing or calling subroutines written in another language;
- declarative marshaling – when developers explicitly inform either the library object or the library users in a certain way about how it is necessary to transform and use the data, usually by specifying the data on the arguments and result types, or
- implicit marshaling – when the corresponding part of the program code or the corresponding declaration is (mostly) automatically generated, so that the programmer does not have to worry about it.

It is clear that implicit marshaling is the most desirable form. Therefore, when designing the interface for combining Wafl programs with classes and subroutines written in C/C++/ASM, one of the main goals was to provide the implicit marshaling in order to facilitate the work of programmers.

The variadic template technique [5] has been applied, to allow functions with different numbers and types of arguments to be automatically registered appropriately. Moreover, most of this work is done in the module compilation phase. When a program written in Wafl uses a module written in C++, the validation of the number and types of arguments is automatically performed prior to the binding. Also, if necessary, the inline function that performs the marshaling is performed automatically. Implicit marshaling simplifies the combining of Wafl and C/C++/ASM and makes the binding safer, while achieving at least the same efficiency as

explicit data transformation.

```
int string_distance( const std::string& s1, const std::string& s2 );
void LibraryImplementation::InitLibrary() {
   *this << "slib"
         << REG_FN( "dist", string_distance,
                    "Compute two strings edit-distance." )
         << ... ;
}
```

Program code 1. Example of C++ module *slib*, with single exported function. Function *string_distance* is declared and implicitly registered with its appropriate type. All marshaling details are implicit.

After building the module (*libwslib.so* for Linux or *libwslib.dll* for Windows), the library modul and function defined in it are used similarly to the libraries written in Wafl (see Program code 2).

```
... slib::dist(...) ...
where {
   slib = extern library 'slib';
}
```

Program code 2. Example of using *slib* module and *dist* function in Wafl.

2.3 Parallelization of Wafl Programs

Parallelization of scripting languages is generally simpler than the parallelization of compiled languages, because it can be done at a slightly higher level of abstraction. In certain cases, it is even possible to completely ignore some complex aspects of parallelization, such as the explicit concern of locking or isolating data or parts of a program. This is especially the case when it comes to functional programming languages where arguments are mostly read-only.

The parallelization of Wafl programs is supported in four different ways, which can even be combined. In all cases, the Wafl parallelization engine decides on some of the important aspects of the parallelization (the number of threads, the number of jobs and others):

1. In its most basic form, parallelization is based on the explicit use of parallel versions of some functions.
2. Another possibility is to declare a part of the program in which parallelization should be supported, by enclosing the parallelized expression by function *parallel*. In this case, all functions contained (directly or indirectly) in the enclosed expression are candidates for parallelization.
3. If only a function name is enclosed by the function *parallel*, then the function call and its definition are candidates to be parallelized.
4. The different aspects of the parallelization can be configured by service configuration, including command line options.

```
... map_par(...)             // (1) using a parallel function map_par
... parallel(...map(...)...) // (2) parallelization of enclosed code
... parallel(map)(...)       // (3) parallelization of enclosed function
```

Program code 3. Example of parallelizations of Wafl code.

The Wafl library includes parallel implementations of many functions. The most important of these are: *map*, *reduce* and *filter*. The parallel implementations have the "_par" suffix (see Program code 3). When implicit parallelization is used, then Wafl parallelization engine decides if a sequential or a parallel version will be used.

3. Results and Conclusions

The applied concepts have been tested on several examples and used in several research projects (for example, for mass computation of edit-distances between protein sequences). Table 1 presents the program efficiency comparison for Mandelbrot set benchmark [6].

Results [s]	C	Python	Python	Python parallel	Wafl	Wafl parallel	Wafl/C	Wafl/C parallel
Mandelbrot	2.36	157.19	157.19	22.27	70.04	11.37	2.38	0.24

Table 1. Mandelbrot set benchmark result, with 400x160 points and 60,000 iterations.
Measured on PC with single CPU with 6 cores and 12 threads.
Single point computation is implemented in C (last two columns).

Our results justify and motivate further work on the development of techniques for implicit parallelization and advanced integration of functional scripting languages and C/C++/ASM. In many cases, the development and especially the parallelization of Wafl programs are much faster than for C/C++/ASM programs. If certain parts of a program are implemented in C/C++/ASM and integrated in the Wafl main program, the efficiency level of C/C++/ASM programs is achievable.

The presented techniques are already in use in bioinformatics research at the Faculty of Mathematics.

References

Breshears C, The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications. O'Reilly Media, Inc., 2009.

Scott ML, Programming Language Pragmatics, 2nd ed. Morgan Kaufmann, 2006.

Malkov S, Customizing a Functional Programming Language for Web Development. Computer Languages, Systems & Structures 36(4):345-351, 2010.

Python 3.10.4 documentation, Extending Python with C or C++, https://docs.python.org/3/ extending/extending.html

Lippman SB, Lajoie J, Moo BE, C++ Primer, 5th ed. Addison Wesley, 2012.

Baker G, Mandelbrot Language Shootout Results, https://ggbaker.ca/prog-langs/mandel/ results.html