Resource and Runtime Efficiency for Multi Alorithmic Fibonacci Algorithm

ABSTRACT

This investigation explores and compares runtime efficiency and resource consumption for both recursive and dynamic programming across several different programming languages that abstract widely differing architectural elements. Utilizing the Fibonacci algorithm, we show that for the algorithms investigated, compiled languages demonstrated measurably better runtime efficiency than the in-ter-preted languages studied. An example is the compiled Go language. Compared to Python, which is in-ter-preted, Go was on av-erage, 1267.44 percent faaster in execution for the recursive algorithm, but 1395.18 percent slower for the dynamic algorithm. At the same time, the results indicated that the compiled languages required more computational resource in pursuit of this faster execution. Looking again at Go compared to Python for the recursive algorithm, Go utilized, on average, 100.02 percent of available processor resource versus 97.67 percent for Python. As for the dynamic algorithm, Go utilized, on average, 101.92 percent of the available processor resource, and Python utilized, on average, 94.92 percent. These results are interesting, taking into account lower machine instruction counts for the compiled languages. A full comparison of all studied languages is presented, the potential factors behind the results analyzed and the possible ramifications for actual use discussed.

1. INTRODUCTION

Algorithmic efficiency has become hugely important due to the need to analyze massive data sets generated by cloud computing and the Internet of Things. While making algorithms more succinct and comprehensive, recursion can also be highly inefficient when applied to these data sets. As an alternative to recursion in many applications, dynamic programming techniques can significantly improve runtime efficiency. Although runtime efficiency has been widely studied for specific problem applications, less attention has been given to the relationship of language and underlying architecture to a broader measure of efficiency that includes both runtime and resource consumption.

Different programming languages are utilized for different purposes, which leads to the question of when to use one language over another? Compiled languages utilize a compiler to take the whole program as input and compile it only once. They generally execute faster than interpreted languages and take up more memory to create the object code as output [1]. Interpreted languages utilize an interpreter, which reads in a single line of code at a time. Because the syntax tree is processed directly to evaluate or execute statements, some of the code may be processed over and over again, resulting in slower execution for interpreted languages [1]. Some common examples of compiled languages include C and Go. Common examples of interpreted languages include Python and Perl.

This paper presents comparisons between several different programming languages, to include interpreted and compiled, and analyzes their performance efficiencies. As a way to make Python more comparable to the compiled language of C, ways of code optimization were explored. As a result, a version of inline C and a Python implementation with a decorator function to store the results needed later for computation were tested.

2. BACKGROUND

In this section, there will be a list of several key concepts and formulas relevant to this research. While this work focuses mainly on C and Python, the generalizations mentioned can be applied to the general concept of compiled languages as opposed to interpreted languages. For this research, C and Go were chosen as the compiled languages, and Perl and Python were chosen for the interpreted languages.

2.1 Programming Languages

We utilized four languages in our study:

1. C is a compiled language created in 1972 at Bell Labs for UNIX system implementation [1]. C is the basis of the programming languages of Java and C++. It is often chosen when speed is a priority for inputs consisting of large data sets [2]. Although it isn't the most simple language to develop with, it is a major player in High Performance Computing (HPC) because of its efficiency in performance [3].

2. Go is a relatively new programming language created by Google. Go is useful for systems programming and scalable network servers. It is a close second behind C when it comes to speed, and is a

relatively simple language to develop with [3]. It is based upon C's implementation; however, it was developed with a focus on a simpler design for the programmer [3].

3. Python is an in-ter-preted lang-uage created in 1991 [4]. It has recently made a large presence in the HPC world through its pack-ages, such as NumPy, SciPy, and scikit-learn; with these packages, Python applications are optmized to take full advantage of the present architecture [4]. With such a presence in HPC, it is clear that Python is here to stay.

4. Perl is a common scripting language that utilizes an interpreter. Like Python, it too, has a large variety of libraries to pull from. Perl is qucker and more efficient than Python in term of input/output operations [5]. In addition, Perl is a useful language due to it having a large number of parallel computing modules available.

2.2 Decorated Python

As a way to make Python more comparable to the execution speed of C, a decorator function with a wrapper was implemented. Decorator functions are more commonly utilized for tracing, locking, or logging [6]. However, a decorator function combined with a Python wrapper can also be created as a way to make a program more dynamic by having it remember the results needed later for computation. This allowed its performance to be much quicker than that of C in terms of execution time.

2.3 Inline C

A method to optimize C was also explored utilizing the keyword ”inline” before the function call. Inline is a useful tool with smaller functions that will be called multiple times in order to reduce function overhead. Utilizing the inline keyword reduces function call overhead by replacing the actual call with the contents of the function itself. Because of this replacement of the function call with the function contents, it is not very useful with multiple recursive calls as there will be a tradeoff in terms of instruction count and efficiency.

3. EXPERIMENTAL METHODOLOGY

In the following subsections, the different aspects of this study will be described.

3.1 Environment

In this subsection, the elements used to set up the environment will be described. We used an Intel NUC NUC5CPYH with ubuntu 16.04 installed, a gcc version of 5.4.0, and python 2.7. We installed the other necessary languages for the experiment, so that the NUC had Perl, Python, C, and Go installed. We also installed perf, a resource monitoring utility, so that we could monitor the resource consumption of the different languages.

3.2 Execution

The algorithm used for testing was the Fibonacci algorithm gathered from Rosetta Code [8]. Rosetta Code is a repository which contains different algorithms with many programming languages to choose from. A recursive and dynamic version of the Fibonacci algorithm was used from this site for the testing algorithm. Fibonacci values of 20, 30, 40, and 50 were used as function parameters for testing purposes.

3.3 Analysis

A profile monitoring resource was used to monitor the resource consumption of the algorithms in order to collect data. Perf is a sample based Linux profiler which monitors Linux perf events [9]. It was utilized for every trial, testing task-clock, CPU-cycles, instruction count, the number of CPUs utilized, clock rate, instructions per second, elapsed time, page-faults, cache-misses, and percent of all cache references. The time command was also used to calculate the time sum consisting of user and system time. Speedup was calculated using the execution times between the different languages.

4. RESULTS

The results for this section will be split up according to algorithm results, programming language results, and optimized programming language results. The optimized language results will include the inline C and decorated Python result explanations.

4.1 Algorithm Results

The overall runtime for the recursive algorithms were larger than the dynamic algorithm runtimes. The overall trend for the recursive algorithms included an increase in resource consumption for the Fibonacci numbers of 30 and 40, whereas the dynamic algorithm results trended mostly linearly. Because of the characteristics of recursive algorithms, it is expected for the runtimes to be slower as there is higher overhead from the call stack being used so heavily. Furthermore, programs are bounded by physical memory, so it is likely that Perl, Python, and Go reached their limit. Otherwise, as the compiler was setting up the activation records, it was trying to do something fancy with the algorithm; thus, causing issues for the runtimes. The dynamic algorithm did not have these issues as there is little overhead. The CPU-cycles may have been reported incorrectly as perf is sample based, and does not count every cycle. It is likely the programs ran too fast and perf didnâĂŹt catch all of the CPU-cycles. In addition, the task-clock and instruction count results can be explained by the higher overhead. For larger Fibonacci numbers, it became unfeasible to calculate the resources as the stack grew too large for the recursive algorithm. The dynamic algorithms were more linear as the resources needed to calculate the larger Fibonacci numbers became higher, because more resources are necessary to deal with larger input values.

4.2 Language Results

The runtimes of Python and Perl appear to be essentially infinite near the Fibonacci number of 30 with the result that no data could be collected for higher numbers. Perl showed a recursive runtime increase of 99.97 percent up to the Fibonacci number of 40. ItâĂŹs interesting that Go was 99.97 percent slower than C, another compiled language, for the recursive algorithm. Also surprisingly, the dynamic algorithm showed a higher runtime for C and Go, versus Perl and Python rather than the inverse, which was expected. Being that interpreted languages are interpreted one line at a time, they generally execute slower. However, in this experiment, they executed faster for the dynamic algorithm. This inconsistency can be explained by the default optimization levels of each of the programs potentially being different, or issues with the scope of the Fibonacci numbers. In addition, Go had an unusually high resource consumption possibly due to perf not calculating the results properly as its results are sample based. Furthermore, all languages should have increased more for the dynamic algorithmic instruction count, as the overhead increased; however, this was not the case. The task-clock rates also should have increased more, but there was a maximum increase of 3.35 percent for the language of Perl.

4.3 C Versus Python Results

C consistently outperformed Python on task-clock time, CPU-cycles, instruction count, instructions per second, and elapsed time. When it came to speedup calculations, C greatly outperformed Python by at least a factor of 28 over Python for fib of 20-40, with fib of 5 for the intervals. Regarding the memory resources, Python had significantly more page faults than C overall. For both the recursive algorithm, and the dynamic algorithm, C had around 40 page faults, whereas Python had about 800 page faults for both its recursive and dynamic algorithm for all fib numbers evaluated. As for the cache misses, Python also fared worse. Python showed exponential growth for its re-cur-sive algorithm cache miss-es, start-ing at about 300000, where-as C start-ed around 15000 for its cache misses, and also depicted exponential growth. Python also had more cache misses in the dynamic algorithm with about 270000 cache misses, whereas C had about 7800 cache misses for the dynamic algorithm. In addition, the cache missesâĂŹ percent of all cache references decreased substantially for Python from 16 to just above 0 for fib of 20-40; this trend was similar to CâĂŹs performance, but from a starting point of 25 down to 10. A cache miss is when the data requested for is not in the cache memory and requires the program to fetch the data from other cache levels, or main memory. These high percentages for the cache misses as a percent of all cache references can lead to significant delays in execution time.

4.4 Inline C Versus C Results

C remained under 1 millisecond for all Fibonacci numbers tested, but inline C went all the way up to 3247.92 milliseconds. C utilized 858,889,235,221 CPU cycles for Fibonacci of 50, whereas inline C only got up to 6,971,907,740 CPU cycles. Inline C utilized a lot more instructions than C did, being that the idea behind the keyword âĂIJinlineâĂİ reduces the amount of function calls by replacing the function call with the contents of the function itself. Both forms of C had about the same number of CPUs utilized. Interestingly, inline C used 100 percent less GHz for its clock rate compared to C for Fibonacci of 20, but inline C and C grow to about the same clock rate after Fibonacci of 50. Initially, inline C had more than double the number of instructions per second than C; however, as the Fibonacci value increased for the function parameter input, the processor speed for both equated to about the same. Both inline C and C had the same average number of page-faults, cache misses, and percent of all cache refs being cache misses. The one-minute difference between the two include C increasing a bit, and then gradually decreasing, while inline C decreases drastically, increases a bit, and then decreases at a more gradual pace.

4.5 Decorated Python Versus Python Results

Py-thon took long-er for its task clock rates than that of Dec-o-rated Py-thon with Fib of 40 be-ing 91871.376 mill-i-sec-onds for Py-thon, and a-bout 36 mill-i-sec-onds for Decorated Python. Decorated Python used far few-er CPU cy-cles with its high-est a-round 61,000,000, and Python us-ing around 132,298,225,415 cy-cles for Fib of 40. Dec-o-rated Py-thon also used far few-er in-struc-tions, and a small-er clock rate. Dec-o-rated Py-thon did use more in-struc-tions, hav-ing a-round 2,500,000,000,000 in-struc-tions, and Py-thon us-ing a-round 28,000,000. With re-gard to cache, Dec-o-rated Py-thon was costl-ier hav-ing more page faults. How-ev-er, it sig-nif-icant-ly im-proved cache miss-es from al-most 42,500,000 cache miss-es with Py-thon to al-most 290,000 cache misses with Dec-o-rated Py-thon. ItâĂŹs in-ter-est-ing to note that Py-thon had a con-tin-ual de-cline down to al-most 0 per-cent for its cache miss-es as a per-cent of all cache ref-er-en-ces; where-as Dec-o-rated Py-thon re-main-ed con-stant at 15 per-cent for Fibonacci values of 20 through 100. nnnnn took long-er for its task clock rates than that of Dec-o-rated Py-thon with Fib of 40 be-ing 91871.376 mill-i-sec-onds for Py-thon, and a-bout 36 mill-i-sec-onds for Dec-o-rated Py-thon. Dec-o-rated Py-thon used far few-er CPU cy-cles with its high-est a-round 61,000,000, and Py-thon us-ing a-round 132,298,225,415 cy-cles for Fib of 40. Dec-o-rated Py-thon also used far few-er in-struc-tions, and a small-er clock rate. Dec-o-rated Py-thon did use more in-struc-tions, hav-ing a-round 2,500,000,000,000 in-struc-tions, and Py-thon us-ing a-round 28,000,000. With re-gard to cache, Dec-o-rated Py-thon was cost-lier hav-ing more page faults. However, it sig-nif-icant-ly im-proved cache misses from almost 42,500,000 cache misses with Python to almost 290,000 cache misses with Decorated Python. ItâĂŹs interesting to note that Python had a continual decline down to almost 0 percent for its cache misses as a percent of all cache references; whereas Decorated Python remained constant at 15 percent for Fibonacci values of 20 through 100.

4.6 C Versus Decorated Python Results

C was fast-er than Dec-o-rated Py-thon with a speed-up of al-most 30 for Fib-o-nacci of 20, but the two switched places after Fib-o-nacci of 30, with Dec-orated Python ending Fibonacci of 40 almost 90 times faster than C. Dec-o-rated Py-thon had few-er CPU cy-cles at 60,976,899.33 cy-cles, with C us-ing 858,889,235,221. Dec-o-rated Py-thon also out-per-formed C in regard to in-stru-ction count, hav-ing 28,075,491.67 in-struc-tions, and C us-ing 6,169,788,902. Both used a-bout 0.9 CPUs, but C used 2.156 GHz, and Dec-o-rated Py-thon only used 1.666 GHz for Fibonacci of 40. On the other hand, Dec-o-rated Py-thon had at least 810 page faults for every Fibonacci value tested, whereas C had at most 41 page faults. Decorated Python was also costlier for cache misses. However, CâĂŹs cache misses as a percent of all cache references decreased from about 25 percent to 10 percent and Decorated Python maintained 15 percent for every trial.

Field	**Value**
File format	PDF
Page limit	11 pages, **not including references or the optional 1-page Appendix**
Paper size	US Letter 8.5in × 11in
Top margin	1in
Bottom margin	1in
Left margin	0.75in
Right margin	0.75in
Body	2-column, single-spaced
Space between columns	0.25in
Body font	10pt
Abstract font	10pt, italicized
Section heading font	12pt, bold
Subsection heading font	10pt, bold
Caption font	9pt (minimum), bold
References	8pt, no page limit, list all authors' names

Table 1: Formatting guidelines for submission.

Please ensure that you include page numbers with your submission. This makes it easier for the reviewers to refer to different parts of your paper when they provide comments.

Please ensure that your submission has a banner at the top of the title page, similar to this one, which contains the submission number and the notice of confidentiality. If using the template, just replace XXX with your submission number.

4.7 Content

~~**Author List.**~~ Reviewing will be double blind; therefore, please do not include any author names on any submitted documents except in the space provided on the submission form. You must also ensure that the metadata included in the PDF does not give away the authors. If you are improving upon your prior work, refer to your prior work in the third person and include a full citation for the work in the bibliography. For example, if you are building on *your own* prior something like: "While the authors of additionally does W, and is therefore much better." Do NOT omit or anonymize references for blind review. There is one exception to this for your own prior work that appeared in IEEE CAL, workshops without archived proceedings, etc. as discussed later in this document.

Figures and Tables. Ensure that the figures and tables are legible. Please also ensure that you refer to your figures in the main text. Many reviewers print the papers in gray-scale. Therefore, if you use colors for your figures, ensure that the different colors are highly distinguishable in gray-scale.

References. There is no length limit for references. **Each reference must explicitly list all authors of the paper. Papers not meeting this requirement will be rejected.** Authors of NSF proposals should be familiar with this requirement. Knowing all authors of related work will help find the best reviewers. Since there is no length limit for the number of pages used for references, there is no need to save space here.

5. PAPER SUBMISSION INSTRUCTIONS

5.1 Guidelines for Determining Authorship

IEEE guidelines dictate that authorship should be based on a **substantial intellectual contribution**. It is assumed that all authors have had a significant role in the creation of an article that bears their names. In particular, the authorship credit must be reserved only for individuals who have met each of the following conditions:

1. Made a significant intellectual contribution to the theoretical development, system or experimental design, prototype development, and/or the analysis and interpretation of data associated with the work contained in the article;

2. Contributed to drafting the article or reviewing and/or revising it for intellectual content; and

3. Approved the final version of the article as accepted for publication, including references.

A detailed description of the IEEE authorship guidelines and responsibilities is available here. Per these guidelines, it is not acceptable to award *honorary* authorship or *gift* authorship. Please keep these guidelines in mind while determining the author list of your paper.

5.2 Declaring Authors

Declare all the authors of the paper upfront. Addition/removal of authors once the paper is accepted will have to be approved by the program chair, since it potentially undermines the goal of eliminating conflicts for reviewer assignment.

5.3 Areas and Topics

Authors should indicate these areas on the submission form as well as specific topics covered by the paper for optimal reviewer match. If you are unsure whether your paper falls within the scope of HPCA, please check with the program chair – HPCA is a broad, multidisciplinary conference and encourages new topics.

5.4 Declaring Conflicts of Interest

Authors must register all their conflicts on the paper submission site. Conflicts are needed to ensure appropriate assignment of reviewers. If a paper is found to have an undeclared conflict that causes a problem OR if a paper is found to declare false conflicts in order to abuse or "game" the review system, the paper may be rejected.

Pease declare a conflict of interest (COI) with the following people for any author of your paper:

1. Your Ph.D. advisor(s), post-doctoral advisor(s), Ph.D. students, and post-doctoral advisees, forever.

2. Family relations by blood or marriage, or their equivalent, forever (if they might be potential reviewers).

3. People with whom you have collaborated in the last FIVE years, including

    - co-authors of accepted/rejected/pending papers. whom you fund.

4. People (including students) who shared your primary institution(s) in the last FIVE years.

5. Other relationships, such as close personal friendship, that you think might tend to affect your judgment or be seen as doing so by a reasonable person familiar with the relationship.

"Service" collaborations such as co-authoring a report for a professional organization, serving on a program committee, or co-presenting tutorials, do not themselves create a conflict of interest. Co-authoring a paper that is a compendium of various projects with no true collaboration among the projects does not constitute a conflict among the authors of the different projects.

On the other hand, there may be others not covered by the above with whom you believe a COI exists, for example, an ongoing collaboration which has not yet resulted in the creation of a paper or proposal. Please report such COIs; however, you may be asked to justify them. Please be reasonable. For example, you cannot declare a COI with a reviewer just because that reviewer works on topics similar to or related to those in your paper. The PC Chair may contact co-authors to explain a COI whose origin is unclear.

We hope to draw most reviewers from the PC and the ERC, but others from the community may also write reviews. Please declare all your conflicts (not just restricted to the PC and ERC). When in doubt, contact the program chair.

5.5 Concurrent Submissions and Workshops

By submitting a manuscript to HPCA'18, the authors guarantee that the manuscript has not been previously published or accepted for publication in a substantially similar form in any conference, journal, or the archived proceedings of a workshop (e.g., in the ACM digital library) – see exceptions below. The authors also guarantee that no paper that contains significant overlap with the contributions of the submitted paper will be under review for any other conference or journal or an archived proceedings of a workshop during the HPCA'18 review period. Violation of any of these conditions will lead to rejection.

The only exceptions to the above rules are for the authors' own papers in (1) workshops without archived proceedings such as in the ACM digital library (or where the authors chose not to have their paper appear in the archived proceedings), or (2) venues such as IEEE CAL where there is an explicit policy that such publication does not preclude longer conference submissions. In all such cases, the submitted manuscript may ignore the above work to preserve author anonymity. This information must, however, be provided on the submission form – the PC chair will make this information available to reviewers if it becomes necessary to ensure a fair review. As always, if you are in doubt, it is best to contact the program chair.

Finally, we also note that the IEEE Plagiarism Guidelines IEEE Plagiarism Guidelines covers a range of ethical issues concerning the misrepresentation of other works or one's own work.

6. ACKNOWLEDGEMENTS

This document is derived from previous conferences, in particular HPCA 2017. We thank Daniel A. Jimenez, Elvira Teran for their inputs.