Asymptotic Analysis of Parallel Algorithms: an Experimental Approach

Mohammad Ghazanfar
Department of Information Science and Engineering
PES Institute of Technology
Bangalore, India

Viraj Kumar
Department of Computer Science and Engineering
PES University
Bangalore, India
viraj.kumar@pes.edu

Abstract— Asymptotic analysis is a key topic in the undergraduate Computer Science curriculum, because it allows one to predict the degree to which an algorithm can tackle larger problems in the same amount of time, given additional computational resources. In the traditional curriculum, such analysis primarily focuses on serial algorithms, partly for simplicity. The plateauing of single-core performance suggests that this focus is misplaced, and the shift towards multi-core and many-core systems strengthens the case for studying asymptotic behavior in the context of parallel algorithms, despite their greater complexity. In this paper, we advocate an experimental approach to complement theoretical asymptotic analysis for parallel and parallelizable algorithms. Specifically, we develop a simple tool to estimate the asymptotic complexity of an algorithm (serial or parallel) from its implementation. We demonstrate how this tool can be used as a pedagogical aid to shed light on important aspects of parallel algorithm design, with emphasis on the asymptotic behavior of the inherently serial fraction of a parallel algorithm’s running time. This fraction plays a significant role in determining the degree to which parallelism can help the algorithm utilize additional computational resources to tackle larger problems.

Keywords—Algorithms, Asymptotic analysis, Amdahl’s Law, Gustafson-Baris’s Law, Karp-Flatt Metric

I. INTRODUCTION

Asymptotic analysis of an algorithm’s running time is a skill that Computer Science undergraduates are expected to master [1]. The asymptotic running time \( t(n) \) of an algorithm depends on the problem size \( n \), and we usually try to tackle as large a problem as possible within a given time limit. If additional time or computational resources become available, an asymptotic analysis can predict how large \( n \) can grow for \( t(n) \) to stay within the time limit. For instance, if \( t(n) \) grows linearly with \( n \), then with twice the computational resources the algorithm can tackle a problem twice as large, provided it can use the additional resources. Until about a decade ago, the italicized proviso caused very little concern: single-core performance was growing exponentially, and it was easy to exploit. Instructors could easily demonstrate the value of asymptotic thinking using real hardware and straightforward serial algorithms, and students could safely ignore several details of the algorithm and focus only on how its asymptotic running time dictated the size of the input that could be tackled within the time limit. There are at least two reasons why it is harder to make a similar point in today’s era of multi-core and many-core systems. First, “additional computational resources” now means additional processing units, so the demonstration requires a more complex parallel algorithm to exploit this hardware. (We target only multi-core, shared-memory systems here.) Second, the benefits of additional processing units may be limited (e.g., when a significant fraction of the computation is inherently serial) or negated altogether (e.g., when there is substantial overhead due to parallelism), and students must remain aware of such details of the algorithm while performing the analysis.

To help students appreciate the necessity for asymptotic analysis, we extend the study of this topic in our Algorithms course as follows. Students are given implementations of serial (but parallelizable) and parallel algorithms, and must experimentally identify algorithms that can utilize additional processing units to enhance their running times. For serial implementations, students make the ideal-case assumptions of Amdahl’s Law [2] (perfect load-balancing and no additional overhead), but these factors must be considered for parallel implementations. (It is extremely challenging to identify the effects of these factors singly, but their combined effects can be estimated using the Karp-Flatt metric [3]).

As part of this research, we have developed a semi-automated tool to help students conduct such experiments. We describe the workings of this tool next (Section II), and we illustrate its use with an example in Section III. Finally, we discuss related work and extensions in Section IV.

II. A TOOL FOR EXPERIMENTAL ANALYSIS

Consider a Java implementation \( A(T x) \) of an algorithm \( A \), where the input \( x \) is of type \( T \). Our tool asks the user to supply a function \( T \ gen(long n) \), which generates a representative input of size \( n \). (For a worst-case analysis, the function \( \text{gen} \) must return an input which triggers the worst-case behavior of the algorithm, which may be non-trivial to create.) Our tool determines an acceptable range of input sizes \( lo \leq n \leq hi \) such that the running time of \( A(\text{gen}(lo)) \) is sufficiently large to be timed accurately (and consistently), and the running time of \( A(\text{gen}(hi)) \) allows several runs to be performed. Our tool is semi-automated, and the user can adjust parameters as necessary. Next, we identify several distinct input sizes \( lo \leq n_1 < n_2 < \ldots < n_i \leq hi \) that are approximately equally spaced across the acceptable range of input sizes. We then time several runs of \( A(\text{gen}(n_i)) \) until we obtain stable running times, and we record these stabilized time \( t(n_i) \) for each \( i \). Our tool then computes values \( f(n_i) \) for each of a family of “common” functions \{ \( f_j \) \} (e.g., \( 1, \log n, n, n \log n, n^2 \), etc.). Lastly, we normalize the running
times by computing all ratios of the form \( t(n) f(n)/f(n_i) \). If this ratio seems to converge to a positive constant as \( n \) increases, our tool reports that \( t(n) = \Theta(f(n)) \). We now describe how students can use such a tool for their experiments.

Let \( t(n, p) \) denote the running time of the algorithm on an input of size \( n \), with \( p \) processing units. The serial algorithm takes time \( t(n, 1) = s(n) + r(n) \), where \( s(n) \) is the inherently serial part of the computation, and \( r(n) \) is the parallelizable (but not yet parallelized) remainder. The inherently serial fraction of the algorithm’s running time is \( f(n) = s(n)/t(n, 1) \).

As per Amdahl’s Law \([2]\), there is scope for exploiting parallelism as \( n \) increases provided \( f(n) = o(1) \) if \( f(n) = \Omega(1) \), then Amdahl’s Law predicts rapidly diminishing returns for greater degrees of parallelism. Students use our tool to estimate the asymptotic complexity of \( t(n, 1) \). Also, if students identify the parallelizable regions, our tool avoids timing these regions and therefore estimates the asymptotic complexity of \( s(n) \).

Hence, the asymptotic behavior of \( f(n) \) can be determined. A similar analysis is done for parallel algorithms: as per Gustafson-Baris’s Law \([4]\), with the same assumptions as before, \( t(n, p) = s(n) + r(n)/p \), and the scaled speedup of the algorithm is \( k(n) = p + (1 - p)\alpha(n) \), where \( \alpha(n) = s(n)/t(n, p) \). Here, parallelism can be exploited if \( k(n) \) does not decrease as \( p \) increases, which once again holds when \( f(n) = o(1) \). Again, our tool can help determine this.

In reality, the ideal-case assumptions of Amdahl’s and Gustafson-Baris’s Laws will not hold: the work-load will be unevenly split, and there will be overhead due to parallelism. In this case, the Karp-Flatt metric \([3]\) estimates \( f(n) \) as \( \alpha(n) = ((t(n, p)/t(n, 1) - 1)/p)(1 - 1/p) \). Our tool can also estimate the asymptotic behavior of \( \alpha(n) \) experimentally.

III. AN EXAMPLE: ALL-PAIRS SHORTEST PATHS

As a simple illustration, consider the classic Floyd-Warshall algorithm to compute shortest paths between each pair of vertices in a weighted \( n \)-vertex graph. The computationally intensive region of this code is shown in Figure 1. Here it is easy to see that \( t(n, 1) = \Theta(n^3) \). Iterations of the outer-most loop must be performed in serial, but the code within can be parallelized. Our tool permits students to mark these parallelizable regions by invoking the methods startParallel() and endParallel() at the start and end of each such region. When the modified code is run, the tool avoids timing these regions, and thereby estimates the asymptotic complexity of \( s(n) \). We note that the changes needed to the given implementation are minimal, once the student has identified the parallelizable region(s).

```java
for(int k = 0; k < n; k++)
    startParallel(); // pause timer
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        d[i][j][k] = Math.min(d[i][j][k-1],
        d[i][k-1][k-1] + d[k-1][j][k-1]);
endParallel(); // resume timer
```

Figure 1. Floyd-Warshall algorithm: innermost loops.

In the parallel implementation, it is quite easy to balance the work-load across processing units and keep the overhead low. However, we additionally expose students to poor implementations with load imbalances and overhead from handling shared memory poorly.

IV. RELATED WORK AND EXTENSIONS

A variety of software tools have been developed for the Algorithms course. TRAKLA2 \([5]\), for instance, offers students practice problems and guidance for performing asymptotic analysis on a pre-defined set of (primarily serial) algorithms. For analyzing arbitrary algorithms, McGeoch et al. \([6]\) have pioneered the experimental analysis approach for serial algorithms, and similar efforts have been made for parallel algorithms \([7]\). In this preliminary study, we have avoided these sophisticated methods and have deliberately built a simple tool for asymptotic analysis to demonstrate the feasibility of our approach.

Our tool is currently being used as described here in an undergraduate Algorithms course. We have packaged it as an easy-to-use API: one only needs to specify how to generate instances of a given size, how to run instances, and (if executing the algorithm destroys the input instance) how to clone instances. The tool is not infallible, but as a support for the pedagogical approach explained in this paper, we believe it is a useful start. We are currently working on improving the accuracy of the tool.

ACKNOWLEDGMENT

We are grateful to the NSF/IEEE TCCP Early Adopter Program for inspiring and supporting this research.

REFERENCES


