Performance Analysis

Dr.N.Sairam & Dr.R.Seethalakshmi
School of Computing,
SASTRA Univeristy,
Thanjavur-613401.

Contents

Performance Analysis 1
1. Performance Analysis 3
1.1 Characteristics of Algorithm 3
1.2 Asymptotic Analysis 3
1.2.1 Big O notation 3
1.2.1.1 Definition [Big-Oh] 3
1.2.2 Theta Notation 4
1.2.2.1 Definition [Theta] 4
1.2.3 Omega Notation 4
1.2.3.1 Definition [Omega] 4
1.3 Performance Metrics for Parallel Systems 4

1. Performance Analysis

An Algorithm is a step by step procedure to achieve a required result and an algorithm is a sequence of computational steps that transform the input into the output.

1.1 Characteristics of Algorithm

Finiteness **:** An algorithm must always run to completion after a finite number of steps.
Definiteness**:** Each step of an algorithm must be precisely defined.
Input **:** an algorithm has zero or more inputs.
Output **:** an algorithm has one or more outputs.
Effectiveness**:** an algorithm is also generally expected to be effective, in the sense that its Operations must all be significantly basic that they can in principle be done exactly and in a finite length of time. It must be optimized. It should be better than the other algorithms for the same problem.

1.2 Asymptotic Analysis

The efficiency or complexity of the algorithm is nothing but the number of steps executed by the algorithm to achieve the results. In theoretical analysis of algorithms, it is common to estimate their complexity in asymptotic sense, i.e., to estimate the complexity function for reasonably large length of input. It's also easier to predict bounds for the algorithm than it is to predict an exact speed. That is, "at the very fastest,this will be slower than this" and "at the very slowest, this will be at least as fast as this" are much easier statements to make than "this will run exactly at this speed". Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve. It's a line that stays within bounds. Asymptotic notation is a shorthand way to write down and talk about 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed. Big *O* notation, omega notation and theta notation are used to this end.

1.2.1 Big O notation

Big *O* notation is a mathematical notation used to describe the asymptotic behavior of functions. Its purpose is to characterize a function's behavior for large inputs in a simple but rigorous way that enables comparison to other functions. More precisely, it is used to describe an asymptotic upper bound for the magnitude of a function in terms function. It is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

1.2.1.1 Definition [Big-Oh]

The function f(n) = O(g(n)) (read as "f of n is big oh of g of n") if(read as "if and only if") there

exist positive constants c and n_0 such that f(n)<= c.g(n) for all n, n>=n_0.

Using *O* notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. Moreover, it is important to understand that the actual running time depends on the particular input of size *n*. That is, the running time is not really a function of *n*. What we mean when we say "the running time is $O(n^2)$" is that the worst-case running time (which is the function of *n*) is $O(n^2)$, or equivalently, no matter what particular input of size n is chosen for each value of n, the running time on that set of inputs is $O(n^2)$.

1.2.2 Theta Notation

The theta notation is used when the function *f* can be bounded both from above and below by the same function *g*.

1.2.2.1 Definition [Theta]

The function f(n) =Θ (g(n)) (read as "f of n is theta of g of n") if(read as "if and only if") there exist positive constants c1, c2 and n0 such that c1.g(n)<=f(n)<= c2.g(n) for all n, n>=n0.

When we write *f(n) =* Θ*(g(n))*, we are saying that *f* lies between c_1 times the function *g* and c_2 times the function *g* except possibly when *n* is smaller than n_0. Here c_1 and c_2 are positive constants. Thus *g* is both a lower and upper bound on the value of *f* for suitably large *n*. Another way to view the theta notation is that it says *f(n)* is both *(g(n))* and *O(g(n))*.

1.2.3 Omega Notation

Just as *O* notation provides an asymptotic upper bound on a function, Omega notation provides an asymptotic lower bound. Since Omega notation describes a lower bound, when we use it to bind the best-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well.

1.2.3.1 Definition [Omega]

The function f(n) = Ω*(g(n)) (read as "f of n is big oh of g of n") if(read as "if and only if") there exist positive constants c and n_0 such that f(n)>= c.g(n) for all n, n>=n_0.*

1.3 Performance Metrics for Parallel Systems

In Parallel computing paradigm number of metrics is important to evaluate the performance of parallel systems. The **parallel runtime** is the time that elapses from the

moment a parallel computation starts to the moment the last processing element finishes execution. The parallel time is represented by $T_{p.}$ We define overhead function or ***total overhead*** of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol *To*. The total time spent in solving a problem summed over all processing elements is *pTP* . *TS* units of this time are spent performing useful work, and the remainder is overhead. We formally define the ***speedup*** *S* 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* processing elements

Consider the example of parallelizing bubble sort. Assume that a serial version of bubble sort of 105 records takes 150 seconds and a serial quicksort can sort the same list in 30 seconds. If a parallel version of bubble sort, also called odd-even sort, takes 40 seconds on four processing elements, it would appear that the parallel odd-even sort algorithm results in a speedup of 150/40 or 3.75. However, this conclusion is misleading, as in reality the parallel algorithm results in a speedup of 30/40 or 0.75 with respect to the best serial algorithm.

Theoretically, speedup can never exceed the number of processing elements, *p*. If the best sequential algorithm takes *TS* units of time to solve a given problem on a single processing element, then a speedup of *p* can be obtained on *p* processing elements if none of the processing elements spends more than time *TS* /*p*. A speed up greater than *p* is possible only if each processing element spends less than time *TS* /*p* solving the problem. In this case, a single processing element could emulate the *p* processing elements and solve the problem in fewer than *TS* units of time. This is a contradiction because speedup, by definition, is computed with respect to the best sequential algorithm. If *TS* is the serial runtime of the algorithm, then the problem cannot be solved in less than time *TS* on a single processing element.

We define the ***cost*** of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on *p* processing elements. The cost of solving a problem on a single processing element is the execution time of the fastest known sequential algorithm. A parallel system is said to be ***cost-optimal*** if the cost of solving a problem on a parallel computer has the same asymptotic growth (in Q terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of Q (1). Cost is sometimes referred to as ***work*** or ***processor-time product***, and a cost-optimal system is also known as a *pTP* -optimal system.

Consider a method for adding n numbers using p processing elements for $n = 16$ and $p = 4$. In the first step of this algorithm, each processing element locally adds its n/p numbers in time $Q(n/p)$. Now the problem is reduced to adding the p partial sums on p processing elements, which can be done in time $Q(\log p)$. The parallel runtime of this algorithm is

$$T_p = \Theta(n/p + \log p)$$

and its cost is $Q(n + p \log p)$. As long as $n = \Omega(p \log p)$, the cost is $Q(n)$, which is the same as the serial runtime. Hence, this parallel system is cost-optimal. Thus the parallel analysis is performed.