Basic concepts: Measuring performance of a parallel system

1 Basic notation

We will use the following notation in describing the attributes of a problem and its algorithms.

Let n denote the input size. Sometimes we will refer to this as the "problem size". Let ω denote the sequential work that is performed by a *best* sequential[1] algorithm for the underlying problem. Note that ω need not be the same as n. For example, the work involved in sorting n numbers is $O(n \log n)$ using merge sort. Both n and ω describe the attributes of the underlying problem.

The following notation will be used with respect to describing the attributes of a parallel algorithm. We will use:

p to denote the number of processors used by an execution of a parallel algorithm. Sometimes we will refer to this as also the "processor size".

$T(n, p)$ to denote the runtime that the algorithm took to complete on a problem size of n and processor size of p. Therefore, $T(n, 1)$ is the runtime taken by the parallel algorithm to run on a single processor. Note that $T(n, 1)$ need not be necessary the same as ω. In fact, it should be easy to see that $T(n, 1) = \Omega(\omega)$.

2 Performance measures

2.1 Speedup

Speedup is defined as the ratio between the sequential time to the parallel runtime. In other words, it is the factor improvement in runtime achieved by the parallel code. We typically use S to denote speedup. By default, we will use the *best* sequential time as the reference. This is also sometimes called the "*real speedup*". Real speedup on p processors is given by:

$$\text{Real speedup } S = \frac{\omega}{T(n,p)} \tag{1}$$

Alternatively, we can also compute a weaker model of speedup called the "*relative speedup*", which is given by:

$$\text{Relative speedup } S = \frac{T(n,1)}{T(n,p)} \tag{2}$$

The notion of relative speedup is considered weaker than real speedup. For most practical applications, it is desirable to report the real speedup as it is more meaningful in conveying the gains yielded by parallelization against the best competing serial alternative. However, in some

[1]We will use the terms "sequential" and "serial" interchangeably.

practical cases use for relative speedup can be justified — for instance, in cases where there is no existing serial implementation that could scale to a large problem instance being studied, or if the goal is to diagnose scaling bottlenecks of a parallel code.

Henceforth, when we say "speedup" we imply real speedup (unless otherwise explicitly stated).

Typical speedup values are in the range of $[1, p]$ for parallel codes, with a $O(p)$ speedup curve representing *linear speedup*. However, as p increases it may become harder to maintain linearity of speedup due to parallel overheads (Section 2.2) and the presence of inherently sequential parts in the parallel program (Section 3), unless the problem size is allowed to scale up (Section 3.1). For these reasons, sub-linear speedups become the norm for large-scale applications, and maintaining the speedup near linear becomes one of the primary challenges in parallel algorithm design.

Alternatively, there could be applications where it is possible to achieve a *super-linear speedup*. Consider the example of a search operation (i.e., a specific search problem instance) where the best serial algorithm takes the worst-case time ($O(n)$) while the parallel implementation finds the solution in $O(1)$ time. Another common reason for observing super-linear speedup could be the effect of caching. A large problem instance when run on say, a single core on a multi-core system may make heavy use of the main memory during computation. However, as more cores are used, the problem size per core shrinks and the local cache lines are more effectively utilized. This could result in drastic reductions in run-time.

Notice the subtle difference between comparing a parallel algorithm's performance to a serial algorithm vs. comparing the performance of different serial algorithms. In the latter, it is typical to use their worst-case performances across inputs to compare; whereas in the former, we generally keep the input instance fixed and compare the parallel code's performance on that input to the best serial code's performance on the same input. As a result, when we study a parallel code's performance, it is common practice to report the speedup curve and other performance curves for each of the inputs tested separately.

2.2 Parallel overhead

A parallel code typically incurs overheads that are not part of a serial code. This could include mundane tasks such as initializing a communicating group of processes (or threads) or tasks that relate to inter-task communication, synchronization, etc. To measure the parallel overhead, it is necessary to quantify the net *work*[2] done by the parallel code and compare it to the work performed by the sequential code. The *net parallel work* is given by $p \times T(n, p)$. Therefore, the parallel overhead incurred by a parallel algorithm on p processors, denoted by $T_o(n, p)$, is given by:

$$\text{Parallel Overhead } T_o(n,p) \quad = \quad p \times T(n,p) - \omega \tag{3}$$

We note here that the parallel overhead is a function that typically increases linearly (if not superlinearly) with the processor size. One of the primary design goals for a parallel algorithm is to keep this overhead to a minimum.

[2]Think of "work" as the volume of computation (as opposed to the length in time) that is performed by a code.

2.3 Parallel efficiency (or simply, efficiency)

Parallel efficiency (E) is a measure of how well the processors in the system are utilized. It is given by the ratio between the work performed by the best sequential algorithm (ω) and the net work performed by the parallel algorithm ($p \times T(n,p)$).

$$\text{Efficiency } E = \frac{\omega}{pT(n,p)} = \frac{S}{p} \tag{4}$$

Efficiency is a fraction and is often expressed as a percentage. For instance, when we observe a parallel code that achieves say 75% efficiency on a system with 100 processors, it tells us th at the code is effectively utilizing 75 processors to perform useful computation. One of the main design goals in designing parallel algorithms is to design methods that can maintain efficiency close to 100% for as large parallel systems as possible (i.e., as p increases).

Example: For the $O(\lg p)$ time parallel algorithm we discussed in class for summing up p numbers using p processors, the speedup and efficiency are as follows:

$$\begin{aligned} S &= O(\frac{p}{\lg p}) \\ E &= O(\frac{1}{\lg p}) \end{aligned}$$

Exercise: Plot these curves for increasing values of p and observe the trends. What happens to the speedup and efficiency as $p \to \infty$?

One of the key properties of efficiency is that it can be preserved when the number of processors is scaled down. This can be stated and shown by the following lemma.

Lemma 2.1. *Let $E(n,p)$ denote the efficiency of a parallel algorithm when run on a given input of size of n and on p processors. If the same algorithm is run on the same input and on p' processors, where $p' < p$, then $E(n,p') \geq (E(n,p))$.*

Proof. By definition of parallel efficiency (Eqn. 4):

$$E(n,p) = \frac{\omega}{p \times T(n,p)}$$

Therefore,

$$\begin{aligned} E(n,p') &= \frac{\omega}{p' \times T(n,p')} \\ &\geq \frac{\omega}{p' \times \left\lceil \frac{p}{p'} \right\rceil T(n,p)} \\ &= E(n,p) \end{aligned}$$

This is because each processor in the reduced pool of p' processors can be made to simulate a unique batch of $\left\lceil \frac{p}{p'} \right\rceil$ processors from the larger pool of p processors—each batch costing at most $\left\lceil \frac{p}{p'} \right\rceil T(n,p)$ runtime. For instance, if we halve the number of processors ($p' = \frac{p}{2}$), then there should be a way to pair up the workloads from each original processor such that the new parallel run-time at most doubles with p' processors. $\square$

Note that the above simulation based argument provides one way to simulate the pool of p processors using a scaled down pool of p' processors. In other words, it suggests that the execution on a lower number of processors can be done without increasing the fraction of time contributed by the parallel overheads. In practice, the contribution from parallel overheads actually tends to decrease when the system is scaled down, resulting in a higher efficiency. In other words, the efficiency curve tends to be a non-increasing (and more typically, decreasing) curve as more processors are added to the system in most real world applications.

2.4 Speedup vs. efficiency

Given that there are two ways to measure a parallel code's performance, viz. speedup and efficiency, which of the two measures should we use in practice to compare algorithms? First, observe that if the number of processors is kept *fixed*, then speedup and efficiency become equivalent (i.e., a higher speedup implies a better efficiency, and vice versa). Therefore, while comparing any two algorithms, we will need to observe their respective behaviors (i.e., measure speedup and efficiency) on varying number of processors, while keeping the input fixed. Consider an example where there are two different parallel algorithms A_1 and A_2 for a given problem such that:

- Algorithm A_1 yields the highest speedup and that occurs when the processor size is p_1;
- Algorithm A_2 yields the highest efficiency and that occurs when the processor size is p_2, where $p_2 < p_1$.

Scenario 1: Now, let us say we want to solve the same problem on p processors such that $p < p_2 < p_1$. Which algorithm should we pick — A_1 or A_2? The answer should be whichever algorithm solves the problem faster on p processors. But can we analytically determine which of the two algorithms would solve the problem quicker on p processors, i.e., without having to run the two codes explicitly? Yes, in fact it can be argued that the algorithm A_2 will be faster than A_1 on p processors. This can be shown by comparing their efficiencies on p processors. The argument is as follows:

Let E_{A_1} and E_{A_2} denote the efficiency functions of the algorithms A_1 and A_2 respectively. Similarly, we will use T_{A_1} and T_{A_2} to denote their respective runtime functions.

By Lemma 2.1, we know that $E_{A_2}(n,p) \approx E_{A_2}(n,p_2)$. Similarly, we also know that $E_{A_1}(n,p) \approx E_{A_1}(n,p_1)$. However, since the peak efficiency was observed for A_2, their expected behaviors on p processors is likely to be: $E_{A_2}(n,p) > E_{A_1}(n,p)$.

$$\Rightarrow \quad \frac{\omega}{pT_{A_2}(n,p)} > \frac{\omega}{pT_{A_1}(n,p)}$$
$$\Rightarrow \quad T_{A_1}(n,p) > T_{A_2}(n,p)$$

In the above example, efficiency provides us a more direct basis using which we can select algorithms. Note that a similar argument can be made using the speedup function although such an argument would still have to piggyback on the efficiency conservation result of Lemma 2.1. Directly comparing speedups would be helpful only when comparing algorithms on the same number of processors.

Scenario 2: Now, consider the case where prediction of system performance on a scaled *up* system becomes important. Let us say we are managing a multi-user, multi-task data center that has a supercomputer with a large number of processors, p_{max}, such that $p_{max} >> p_1$. Assume that the users are submitting multiple tasks that entail roughly identical work, and that each of those tasks is similar to the $O(n)$ input described under Scenario 1. For sake of simplicity, let us assume that all tasks have already been submitted and are available to the scheduling software, which is responsible of coming up with an execution plan for all the tasks.

Since the peak speedup was observed for algorithm A_1 on p_1 processors, it can be inferred that there is no point running that algorithm (or for that matter, A_2) on a larger processor size than p_1. Therefore, a more realistic use-case here for either of the two algorithms would be to run multiple instances of the algorithm concurrently, each task serving a different user submitted job. In other words, the goal becomes one of maximizing *throughput*, where throughput is defined as the rate at which user tasks are processed using as much of the p_{max} processors concurrently.

$$\text{Throughput} = \text{number of tasks completed per unit time} \tag{5}$$

If we decide to run each task using algorithm A_1 on p_1 processors, then the throughput is:

$$\text{Throughput}_{A_1} = \frac{\left\lceil \frac{p_{max}}{p_1} \right\rceil}{T_{A_1}(n, p_1)} \approx \frac{p_{max}}{p_1 \times T_{A_1}(n, p_1)}$$

Alternatively, if we decide to run each task using algorithm A_2 on p_2 processors, then the throughput is:

$$\text{Throughput}_{A_2} = \frac{\left\lceil \frac{p_{max}}{p_2} \right\rceil}{T_{A_2}(n, p_2)} \approx \frac{p_{max}}{p_2 \times T_{A_2}(n, p_2)}$$

However, since the efficiency of A_2 on p_2 processors is more than the efficiency of A_1 on p_1 processors, $p_2 \times T_{A_2}(n, p_2) < p_1 \times T_{A_1}(n, p_1)$.

$$\Rightarrow \text{Throughput}_{A_2} > \text{Throughput}_{A_1}$$

Therefore, it would be more prudent to run multiple instances of algorithm A_2 on the parallel system as that would yield in a higher throughput. Intuitively, this argument makes sense because algorithm A_2 makes better utilization of the resources it is provided than A_1, at least on system sizes that contain p_2 processors or less. Therefore, running A_2 on p_2 processors will be more optimal from a throughput perspective.

Note that there is a caveat here in practice, that the user is willing to wait for $T_{A_2}(n, p_2)$ for completion of each task. If this time exceeds the budget then settling for a slightly larger number of processors than p_2 at the expense of ideal efficiency would perhaps make more sense. It is for this reason that maintaining a high efficiency for as large a processor size as possible becomes an important design goal during parallel algorithm design.

3 Amdahl's law

Speedups are important from the perspective of reducing the time-to-solution. A good parallel algorithm would provide increasing speedups as more processors are added to the system. However, this trend *cannot* continue indefinitely. This is shown by Amdahl's law, which observes that every parallel program contains within it an inherently sequential portion and a parallel portion. Let ω_s and ω_p denote the serial and parallel fractions of the total work performed by a parallel program on a specific input (i.e., $\omega_s + \omega_p = 1$). On p processors, the time is then divided up as follows:

$$T(n,p) = \omega_s + \frac{\omega_p}{p}$$

$$\Rightarrow \text{maximum achievable speedup on } p \text{ processors} = \frac{\omega_s + \omega_p}{\omega_s + \frac{\omega_p}{p}} = \frac{1}{\omega_s + \frac{1-\omega_s}{p}} \quad (6)$$

The above equation is called *Amdahl's law* and it provides a bound the maximum achievable speedup for any parallel program on p processors. Intuitively, it implies that as the sytem size is increased it is only the parallel portion that benefits, while the serial portion starts to dominate the overall run-time, thereby limiting the speedup that can be achieved.

Amdahl's law holds when the input size is fixed and the processor size is increased. The study of a parallel program on increasing processor size keeping the input size fixed is sometimes referred to as *strong scaling*.

Example: Consider a serial program that can be 90% parallelized — i.e., $\omega_p = 0.9$ and $\omega_s = 0.1$. And let the time to run the serial program (in serial) is 100 seconds (i.e., $\omega = 100$). Then by Amdahl's law, the maximum achievable speedup using p processors is $\frac{100}{10+\frac{90}{p}}$. The above suggests that there is little merit in using more than 10 processors to solve this problem as no practically no improvement in speedup can be achieved. Furthermore, even if the number of processors is doubled from 5 to 10, the resulting speedup only goes up marginally, from 3.57x to 5.26x. Based on this observation, one can argue that there is little merit in using more than even 5 processors for this problem.

3.1 Gustafson's law

In 1988, John Gustafson observed that Amdahl's law could potentially pose a mental barrier among parallel programmers as it can be used as an excuse for not scaling up to large processor sizes [1]. In his paper, he argues that it is indeed possible to achieve near linear speedups on large-scale systems (i.e., with thousands of processors). The key is to scale up the problem/input size along with the processor size so as to keep the time to solution as flat as possible (e.g., double n while doubling p). This would result in the scaling up of local computation relative to the overheads introduced due to increased parallelism.

$$\Rightarrow \text{maximum achievable } \textit{scaled} \text{ speedup on } p \text{ processors} = \frac{\omega_s + p \times \omega_p}{\omega_s + \omega_p} \quad (7)$$

This is a simple and yet important result and reflects the general way we use parallel systems in the real world. Please refer to the paper for further details. The study of parallel systems by increasing both processor and problem sizes in tandem is referred to as *weak scaling*.

4 Isoefficiency

While conducting weak scaling studies, it is important to estimate how much the serial work (ω) should be increased to, as we increase the number of processors by a constant factor. The goal of weak scaling studies is to maintain efficiency by solving a larger problem in roughly the same alloted run-time. To facilitate calculating the rate at which the input work should increase, the *isoefficiency* function was introduced, which can be derived as follows:

Recall from Eqn 3 the parallel overhead can be measured as follows:

$$\begin{aligned} T_o(n,p) = p \times T(n,p) - \omega \\ \Rightarrow \quad T(n,p) &= \frac{T_o(n,p)+\omega}{p} \\ \Rightarrow \quad \text{Speedup } S &= \frac{p\omega}{T_o(n,p)+\omega} \\ \Rightarrow \quad \text{Efficiency } E &= \frac{\omega}{T_o(n,p)+\omega} \\ \Rightarrow \quad E &= \frac{1}{1+\frac{T_o(n,p)}{\omega}} \end{aligned} \tag{8}$$

Eqn 8 indicates the following: if ω is fixed and p is increased, E degrades. Alternatively, if p is fixed and w is increased, E could be improved. A "scalable" parallel system is one that maintains efficiency as the system size is increased. This can be achieved by increasing the input work ω. However, at what rate should ω be increased to maintain efficiency? To keep efficiency fixed in Eqn 8, we need to keep the ratio between $T_o(n,p)$ and ω fixed. This can be seen by rearranging Eqn 8 as follows:

$$\begin{aligned} E = \frac{1}{1+\frac{T_o(n,p)}{\omega}} \\ \Rightarrow \quad \omega + T_o(n,p) &= \frac{\omega}{E} \\ \Rightarrow \quad T_o(n,p) &= \frac{\omega(1-E)}{E} \\ \Rightarrow \quad \omega &= \frac{E}{1-E}T_o(n,p) \\ \Rightarrow \quad \omega &= K \times T_o(n,p) \end{aligned} \tag{9}$$

where K is the isoefficiency constant. Eqn 9 is called the *IsoEfficiency* function. The function can be used to calculate the appropriate value of ω that will help maintain efficiency at a target (predetermined) value E. Note that $T_o(n,p)$ can be pre-determined as well using the parallel and serial time complexities.

Example: Consider the problem of summing p numbers on p processors (i.e., $n = p$); The serial work for this problem is $\omega = \Theta(p)$. As discussed in class, we can compute the sum in $\Theta(\lg p)$ time in parallel. Let us now derive the isoefficiency function for this parallel algorithm.

$$\begin{aligned} T_o(n,p) &= p(\lg p - 1) \\ \Rightarrow \quad \omega &= K \times p(\lg p - 1) \end{aligned}$$

where $K = \frac{E}{1-E}$ is the isoefficiency constant. Now, let us say we want to double the number of processors. By what factor should the serial work increase by in order to maintain efficiency? If we use the isoefficiency function, then it should be easy to see that the new serial work assigned for computation on $2p$ processors should grow by a factor of $\frac{2\lg p}{\lg p - 1}$ in order to maintain efficiency. (Proof left as exercise.)
What this implies in practice is as follows. For example, if the number of processors is increased from 8 to 16, then the above equation implies that the serial work (in this case, equal to the input array size) should grow by a factor of 3x. On the other hand, if the number of processors is increased from 1024 to 2048, then growing the serial work by a factor of 2.2x would suffice.

Exercise: Provide the parallel speedup, efficiency and isoefficiency functions for the parallel algorithm used to sum n numbers on p processors (assuming $n >> p$, n is divisible by p, and p is a power of 2).

[1] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. 01225.