Scalability Issues : Amdahl's Law

- Amdahl's law is the simplest form of a scaling law. The underlying assumption is that the performance of the parallel code scales with the number of threads. This is unrealistic, as we will discuss later, but does provide a basic starting point. If we assume that **S** represents the time spent in serial code that cannot be parallelized and **P** represents the time spent in code that can be parallelized, then the runtime of the serial application is as follows:

  Runtime = S + P

- The runtime of a parallel version of the application that used N processors would take the following: Using Parallelism to Improve the Performance of a Single Task

$$\text{Runtime} = S + \frac{P}{N}$$

- It is probably easiest to see the scaling diagrammatically. In Figure 3.7, we represent the runtime of the serial portion of the code and the portion of the code that can be made to run in parallel as rectangles:

Fig: Single Threaded run time.

- If we use two threads for the parallel portion of the code, then the runtime of that part of the code will halve, and Figure below represents the resulting processor activity.

Fig:Runtime with two threads

- If we were to use four threads to run this code, then the resulting processor activity would resemble Figure below.

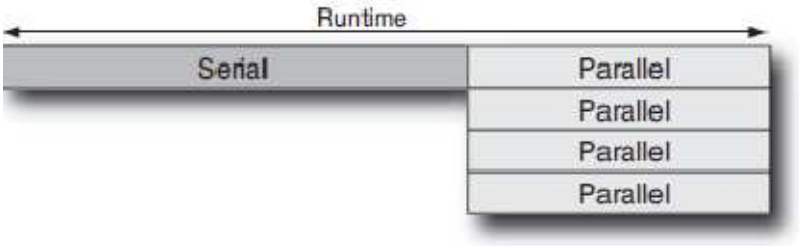

Fig: Run time with four threads

- There are a couple of things that follow from Amdahl's law. As the processor count increases, performance becomes dominated by the serial portion of the application.
- Another observation is that there are diminishing returns as the number of threads increases: At some point adding more threads does not make a discernible difference to the total runtime.
- These two observations are probably best illustrated using the chart in Figure below.

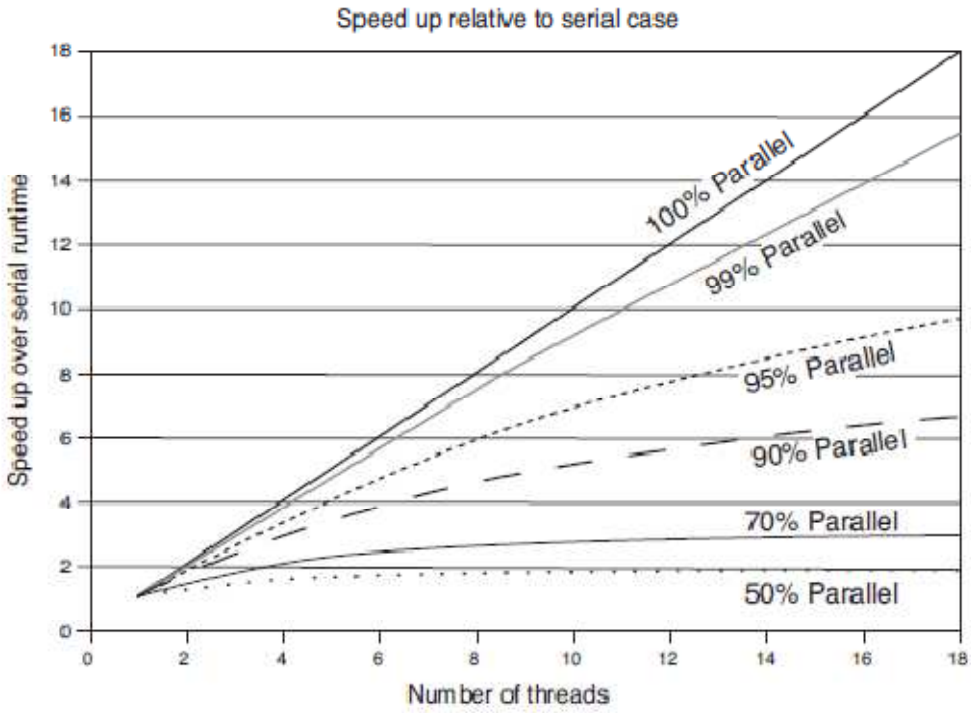

Fig: Scaling with diminishing parallel regions

- Another way of using Amdahl's law, and that is to look at how many threads
  an application can scale to given the amount of time it spends in code that can be parallelized.

- the runtime of an application that has a proportion P of parallelizable code and S of serial code and that is run with N threads is as follows:

$$\text{Runtime}_N = S + \frac{P}{N}$$

- The optimal runtime, when there are an infinite number of threads, is S. So, a runtime within T percent of the optimal would be as follows:

$$\text{Acceptable runtime} = S*(1+T)$$

- We can compare the acceptable runtime with the runtime with N threads:

$$S*(1+T) = \left(S + \frac{P}{N}\right)$$

- We can then rearrange and solve for N to get the following relationship for N:

$$N = \frac{P}{ST} = \frac{P}{(1-P)T}$$

- If Amdahl's law were the only constraint to scaling, then it is apparent that there is little benefit to using huge thread counts on any but the most embarrassingly parallel applications.
- If performance is measured as throughput (or the amount of work done), it is probable that for a system capable of running many threads, those threads may be better allocated to a number of processes rather than all being utilized by a single process.

How Synchronization Costs Reduce Scaling

- There are overhead costs associated with parallelizing applications. These are associated with making the code run in parallel, with managing all the threads, and with the communication between threads. When there are multiple threads cooperating to solve a problem, there is a communication cost between all the thread.

- We can denote this synchronization cost as some function F(N), since it will increase as the number of threads increases. In the best case, F(N) would be a constant, indicating that the cost of synchronization does not change as the number of threads increases.

- In the worst case, it could be linear or even exponential with the number threads. A fair estimate for the cost might be that it is proportional to the logarithm of the number of threads **(F(N)=K*ln(N));** this is relatively

easy to argue for since the logarithm represents the cost of communication if those threads communicated using a balanced tree. Taking this approximation, then the cost of scaling to N threads would be as follows:

$$\text{Runtime} = S + \frac{P}{N} + K \ln(N)$$

- The value of K would be some constant that represents the communication latency between two threads together with the number of times a synchronization point is encountered (assuming that the number of synchronization points for a particular application and workload is a constant). K will be proportional to memory latency for those systems that communicate through memory, or perhaps cache latency if all the communicating threads share a common level of cache.

- It is relatively straightforward to calculate the point at which this will happen:

$$\frac{d\ \text{runtime}}{dN} = \frac{-P}{N^2} + \frac{K}{N}$$

- Solving this for N indicates that the minimal value for the runtime occurs when

$$N = \frac{P}{K}$$

- This tells us that the number of threads that a code can scale to is proportional to the ratio of the amount of work that can be parallelized and the cost of synchronization. So, the scaling of the application can be increased either by making more of the code run in parallel (increasing the value of P) or by reducing the synchronization costs (reducing the value of K). Alternatively, if the number of threads is held constant, then reducing the
synchronization cost (making K smaller) will enable smaller sections of code to be made parallel (P can also be made smaller).