Parallel Programming Principle and Practice

Performance analysis

Parallel algorithm design and performance analysis

Performance analysis

Outline

- Performance analysis
  - Speedup
    - ✓ Linear speedup
    - ✓ Super-linear speedup
    - ✓ Sub-linear speedup
  - Efficiency
  - Scalability
    - ✓ Asymptotic analysis
      - Strong scalability
        - Amdahl's law
      - Weak scalability
        - Gustafson's law
      - Isoefficiency

Performance analysis

Speedup

Speedup

- Definition
  - In computer architecture, speedup is a number that measures the relative performance of two systems processing the same problem
  - More technically, it is the improvement in speed of execution of a task executed on two similar architectures with different resources

  - The notion of speedup was established by Amdahl's law, which was particularly focused on parallel processing
  - However, speedup can be used more generally to show the effect on performance after any resource enhancement

Speedup

- Speedup can be defined for two different types of quantities:

latency and ***throughput***

- ***Latency*** of an architecture is the reciprocal of the execution speed of a task:

$$L = \frac{1}{v} = \frac{T}{W},$$

where

- v is the execution speed of the task;
- T is the execution time of the task;
- W is the execution workload of the task.

Latency is often measured in seconds per unit of execution workload. cycles per instruction (CPI), is an example of latency.

Speedup

- Speedup can be defined for two different types of quantities:

latency and ***throughput***

- ***Throughput*** *of an architecture is the execution rate of a task:*

$$Q = \rho v A = \frac{\rho A W}{T} = \frac{\rho A}{L},$$

where

- ρ is the execution density (e.g., the number of stages in an instruction pipeline for a pipelined architecture);
- A is the execution capacity (e.g., the number of processors for a parallel architecture).

Throughput is often measured in units of execution workload per second. An example of throughput is instructions per cycle (IPC)

Speedup

- Speedup in latency :
  - Speedup in latency is defined by the following formula:

$$S_{\text{latency}} = \frac{L_1}{L_2} = \frac{T_1 W_2}{T_2 W_1},$$

where

- S_{latency} is the speedup in latency of the architecture 2 with respect to the architecture 1;
- L_1 is the latency of the architecture 1;
- L_2 is the latency of the architecture 2.

Speedup in latency can be predicted from Amdahl's law or Gustafson's law , which will be introduced later.

Speedup

- Speedup in throughput :
  - Speedup in throughput is defined by the following formula:

$$S_{\text{throughput}} = \frac{Q_2}{Q_1} = \frac{\rho_2 A_2 T_1 W_2}{\rho_1 A_1 T_2 W_1} = \frac{\rho_2 A_2}{\rho_1 A_1} S_{\text{latency}},$$

where

- $S_{\text{throughput}}$ is the speedup in throughput of the architecture 2 with respect to the architecture 1;
- Q_1 is the throughput of the architecture 1;
- Q_2 is the throughput of the architecture 2.

Speedup: Example

- Using execution times:
  - We are testing the effectiveness of a branch predictor on the execution of a program
  - First, we execute the program with the standard branch predictor on the processor, which yields an execution time of **2.25** seconds
  - Next, we execute the program with our modified (and hopefully improved) branch predictor on the same processor, which produces an execution time of **1.50** seconds
  - In both cases the execution workload is the same
  - Using our speedup formula, we know

$$S_{\text{latency}} = \frac{L_{\text{old}}}{L_{\text{new}}} = \frac{2.25\ \text{s}}{1.50\ \text{s}} = 1.5.$$

Our new branch predictor has provided a 1.5x speedup over the original

Speedup:Example

- Using cycles per instruction:
  - We can also measure speedup in cycles per instruction (CPI) which is a latency.
  - First, we execute the program with the standard branch predictor, which yields a CPI of 3
  - Next, we execute the program with our modified branch predictor, which yields a CPI of 2
  - In both cases the execution workload is the same
  - Using the speedup formula gives

$$S_{\text{latency}} = \frac{L_{\text{old}}}{L_{\text{new}}} = \frac{3\ \text{CPI}}{2\ \text{CPI}} = 1.5.$$

Our new branch predictor has provided a 1.5x speedup over the original

Linear speedup

- Considering the parallel computing:
  - Let ***S*** be the speedup of execution of a task and ***s*** the number of professors,
  - In ideal situation, ***S*=*s***
  - Linear speedup or ideal speedup is obtained when ***S* = *s***.
    - When running a task with linear speedup, doubling the local speedup doubles the overall speedup
    - As this is ideal, it is considered very good scalability

sub-linear speedup

- However, in real word, it is really hard to achieve the *Linear speedup* or *ideal speedup*.
- When **S<s**, it is called **sub-linear speedup**
- For example, for **p** professors , s=p:

$$\text{Speedup} = \frac{\text{serial time}}{\text{parallel time}} = S(p) \rightarrow p$$

$\boldsymbol{S(p)/s = S(p)/p}$ is typically between 0 and 1.

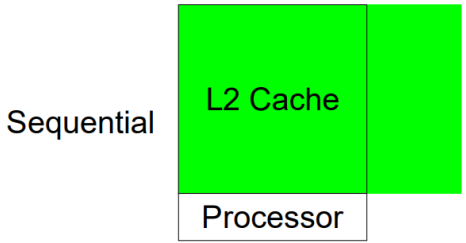
- Programs with linear speedup and programs running on a single processor have an efficiency of 1
- many difficult-to-parallelize programs have efficiency that approaches 0 as the number of processors p = s increases

Super-linear speedup

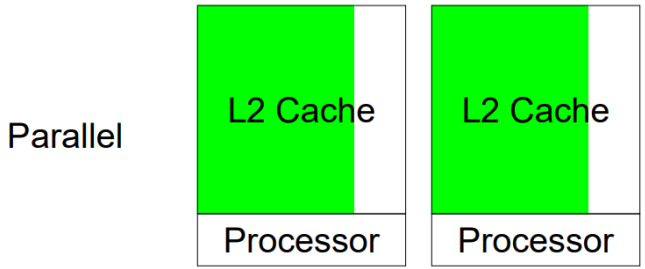
- Sometimes a speedup of more than A when using A processors is observed in parallel computing, which is called **super-linear speedup**
  - Super-linear speedup **rarely happens** and often confuses beginners, who believe the theoretical maximum speedup should be A when A processors are used
  - One possible reason for super-linear speedup in low-level computations is the cache effect resulting from the different memory hierarchies of a modern computer
    - ✓ in parallel computing, not only do the numbers of processors change, but so does the size of accumulated caches from different processors
      - With the larger accumulated cache size, more or even all of the working set can fit into caches and the memory access time reduces dramatically, which causes the extra speedup in addition to that from the actual computation

Super-linear speedup

- For example, for 1 professor : the only cache is not enough for data storage in the task , so it need read-data time



- For 2 professor in parallel : two caches have enough space for data storage in the task , so it doesn't need read-data time



Performance analysis

Efficiency

Efficiency

- Efficiency is a metric of the utilization of the resources of the improved system defined as

$$\eta = \frac{S}{s}.$$

  - ✓ S be the speedup of execution of a task
  - ✓ s the number of professors

- For p professors:

$$\text{Speedup} = \frac{\text{serial time}}{\text{parallel time}} = S(p) \to p$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{S(p)}{p} = E(p) \to 1$$

Let T_s denote the serial time, T_p the parallel time, and T_0 the overhead, then: $pT_p = T_s + T_0$.

$$E(p) = \frac{T_s}{pT_p} = \frac{T_s}{T_s + T_0} = \frac{1}{1 + T_0/T_s}$$

The scalability analysis of a parallel algorithm measures its capacity to effectively utilize an increasing number of processors.

Efficiency

relating efficiency to work and overhead

Let W be the problem size.

The overhead T_0 depends on W and p: $T_0 = T_0(W, p)$.

The parallel time equals $T_p = \dfrac{W + T_0(W, p)}{p}$

Speedup $S(p) = \dfrac{W}{T_p} = \dfrac{Wp}{W + T_0(W, p)}$.

Efficiency $E(p) = \dfrac{S(p)}{p} = \dfrac{W}{W + T_0(W, p)} = \dfrac{1}{1 + T_0(W, p)/W}$.

The goal is for $E(p) \to 1$ as $p \to \infty$.

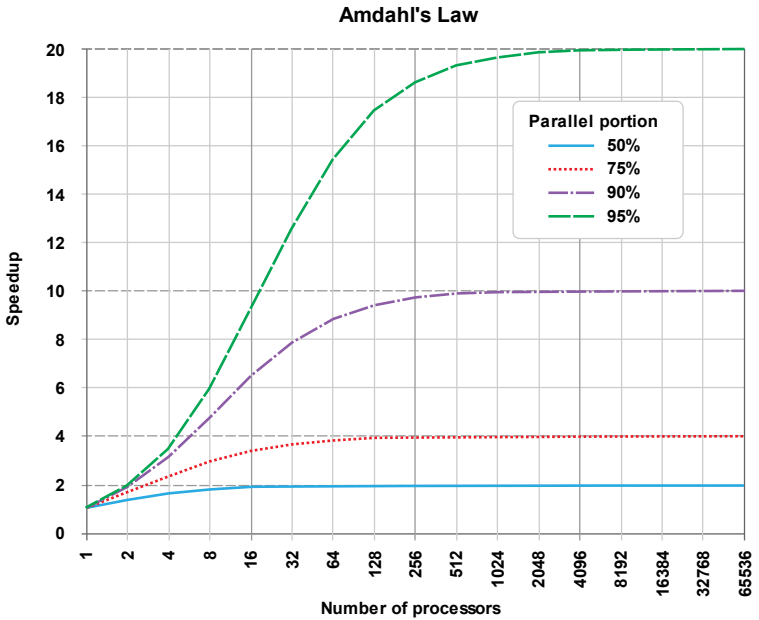
The algorithm scales badly if W must grow exponentially to keep efficiency from dropping. If W needs to grow only moderately to keep the overhead in check, then the algorithm scales well.

Performance analysis

Scalability

Strong scalability: Amdahl's Law

- Named after computer scientist **Gene Amdahl**
  - ✓ Presented at the AFIPS Spring Joint Computer Conference in 1967
- A formula which gives the theoretical speedup in latency of the execution of a task at fixed workload
- Indicates that "**the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used**"
- Usually used for parallel computing to predict the theoretical speedup when using multiple processors



根据阿姆达尔定律，程序执行延迟的理论加速是执行它的处理器数量的函数。加速受到程序串行部分的限制。例如，如果95%的程序可以并行化，则使用并行计算的理论最大加速将是20倍。

Source：https://en.wikipedia.org/wiki/Amdahl%27s_law

Amdahl's Law

- 假设原来在一个系统
  - 执行一个程序需要时间T_{old}
  - 其中某一个部分占的时间百分比为 a
  - 把这一部分的性能提升k倍
  - 即这一部分原来需要的时间为$a*T_{old}$
  - 现在需要的时间变为$(a*T_{old})/k$
  - 则整个系统执行此程序需要的时间变为：$T_{new}=(1-a)T_{old}+(aT_{old})/k=T_{old}[(1-a)+a/k]$
  - 故可得，系统性能提速的倍数为：$S = \frac{1}{(1-a)+a/k}$

Amdahl's Law

Example

- 假设某个系统的某个部分的执行时间占总执行时间的60%, 即 a = 60%
- 这部分性能提升3倍(k=3)
- 则整个系统的性能提升为$\frac{1}{(1-0.6)+0.6/3}$=1.67倍
- 极端化
  - ✓ 将上述部分性能的提升从3倍改为无穷大倍，即 $k = \infty$，这部分能瞬间执行完
  - ✓ 则$S_{\infty} = \frac{1}{1-a} = \frac{1}{1-0.6}$ =2.5，远小于∞

Conclusion

- 即使系统的主要部分(main part)性能提升很多，整个系统的性能提升远远小于此部分的提升
- 当 k 趋近于无穷大时（即假设我们有无穷多个核心），速度提升的上限是 1/(1-a)，即速度提升的上限取决于程序不能被并行计算的部分

Amdahl's Law

- Amdahl's Law
  - The calculation formula and theoretical upper limit of the speedup ratio after parallelization of the serial system are defined
  - How much does improve the performance of one part of a system affecting the whole system
- When part of the system performance is improved, the impact on the performance of the whole system depends on
  - How important is this part
  - How much performance has this part improved

Amdahl's Law

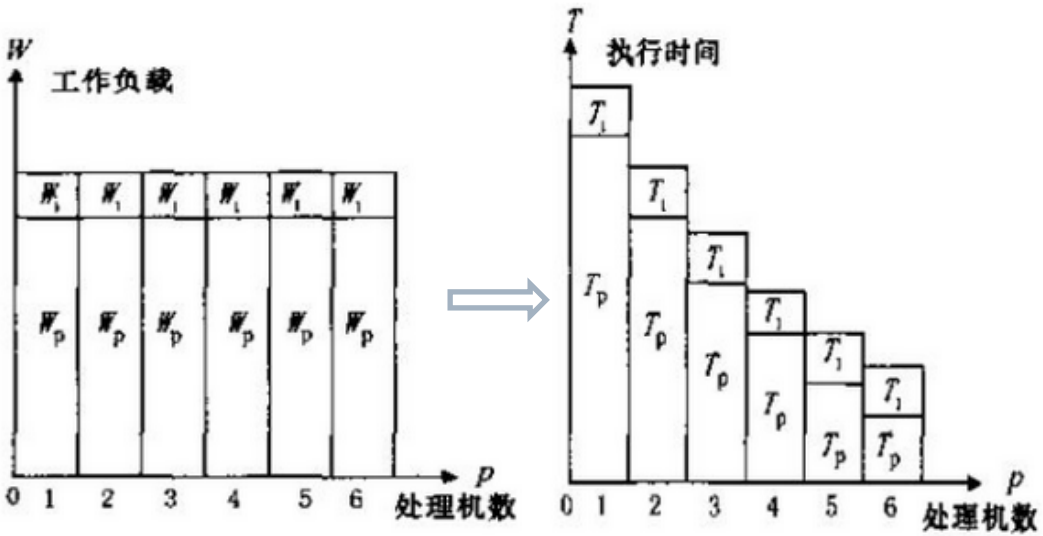
- 假设:
  - 串行比例
    - ✓ f= $\frac{W_s}{W}$
  - 并行比例
    - ✓ 1-f
  - 定义加速比为

$$S = \frac{W_s+Wp}{\frac{W_p}{p}+Ws} = \frac{(Ws+Wp)/W}{(\frac{W_p}{p}+Ws)/W} = \frac{[f+(1-f)]}{f+\frac{1-f}{p}} = \frac{p}{1+f(p-1)} = \frac{1}{f}(p->\infty)$$

可知当处理机数无限增加时，加速比仅为$\frac{1}{f}$

Amdahl's Law

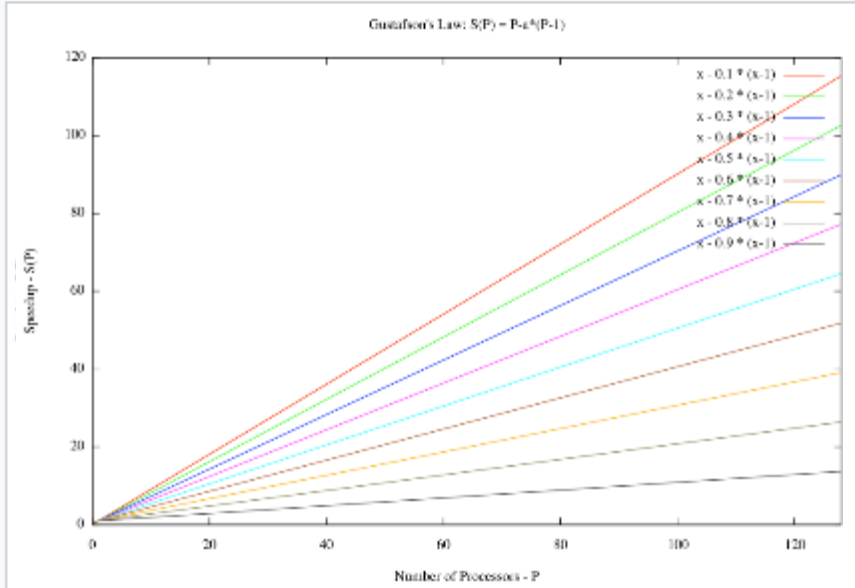
- **Amdahl几何意义**



计算负载固定不变 ⟹ 处理机数增加，执行时间缩短

Weak scalability: Gustafson's law

- Named after computer scientist John L. Gustafson and his colleague Edwin H. Barsis
  - Presented in the article Reevaluating Amdahl's Law in 1988
- Gives the speedup in the execution time of a task that theoretically gains from parallel computing (using a hypothetical run of the task on a single-core machine as the baseline)
- The theoretical "slowdown" of an already parallelized task if running on a serial machine



根据古斯塔夫森定律的演化，理论加速在程序执行延迟的函数中，对于不同的值a。

Source：https://en.wikipedia.org/wiki/Gustafson%27s_law

Gustafson's law

- execution time = Ts + Tp
- Total execution time = Ts + n*Tp
- S(n) = (Ts + n*Tp) / (Ts + Tp)
- F = Ts / (Ts + Tp)

⇒

$$
\begin{aligned}
S(n) &= (Ts + n*Tp) / (Ts + Tp) \\
&= Ts / (Ts + Tp) + n*Tp / (Ts + Tp) \\
&= F + n*(Ts + Tp - Ts)/ (Ts + Tp) \\
&= F + n*[1-Ts/(Ts + Tp)] \\
&= F + n*(1-F) \\
&= F + n - n*F \\
&= n - F(n-1)
\end{aligned}
$$

可见:加速比S(n)与处理机数目n基础成线性关系

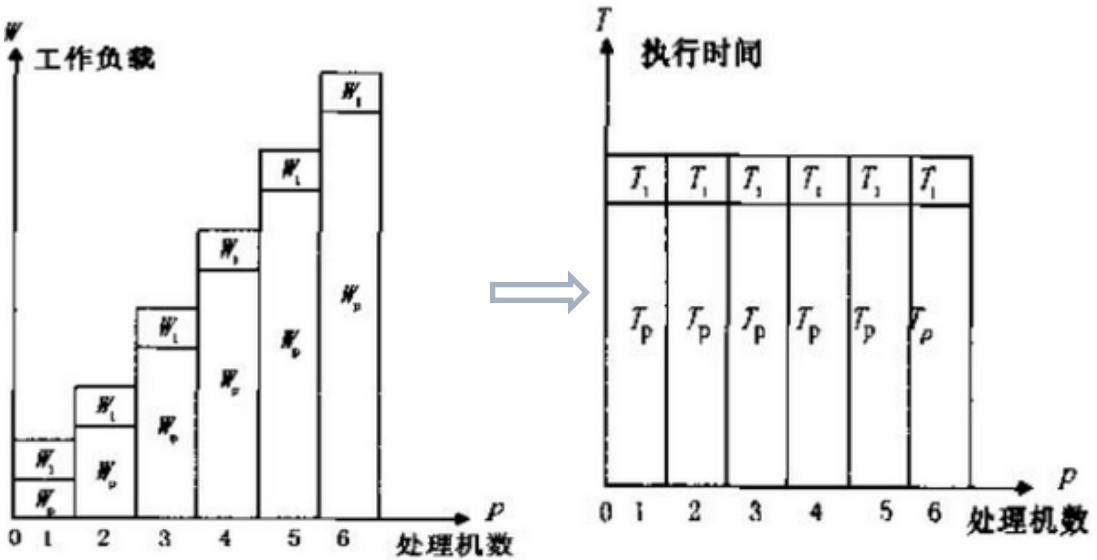
Gustafson's law

- Example
  - Consider that a certain application executes in 220 seconds in 64 processing units
  - 5% of the execution time is spent on sequential computations
  - What is the speedup of the application?

$$S_{(64)} = n - F(n-1)$$
$$= 64 - 0.05(64-1)$$
$$= 64 - 3.15$$
$$= 60.85$$

Gustafson's law

Gustafson's的几何意义



工作量增加，为了保证处理时间不变，需要增加处理器

增加处理器数目，可保证执行时间不变

Isoefficiency (恒等效率)

- A relative relationship between the workload required to keep the efficiency E fixed and the machine size **n** when a parallel algorithm is implemented on a parallel computer
- Assume
  - Workload: W = W(s)
  - Communication overhead: h = h(s,n)
    - ✓ h increases with the increase of s and n
    - ✓ s is the problem size
    - ✓ n is the machine size

Isoefficiency

- Efficiency: $E = \frac{W(s)}{W(s)+h(s,n)}$
  - Given the size of the machine, the cost h grows slower than the workload W
  - For a machine of a certain size, the efficiency increases with the size of the problem
  - Efficiency may remain the same if workload W increases appropriately with machine size
- The constant efficiency functions of general parallel algorithms are polynomial functions of n, eg $O(n^k)$, $k \geq 1$
- The smaller the power of n, the greater the scalability of the parallel system (the system includes a combination of algorithms and structures)

Isoefficiency

- Parallel program execution time: $T_p = (T_1+T_0)/p$
  - T_1 : Total workload serial execution time
  - T_0 : Total communication delay of multiple nodes
  - p : The number of nodes
- Speedup
  - $S = \frac{T1}{Tp}$
  - $E = \frac{S}{p} = \frac{T_1}{pT_p} = \frac{T_1}{T_1+T_0} = \frac{1}{1+\frac{T_0}{T_1}}$
  - $T_1 = W * t_c$
  - W : Total workload calculated by the number of operations
  - t_c : Average execution time for each operation

Isoefficiency

- $E = \frac{1}{1+\frac{T_0}{WT_c}} \Rightarrow \frac{T_0}{W} = t_c(\frac{1-E}{E})$
- $W = \frac{1}{t_c}(\frac{E}{1-E})T_0$
  - E is a fixed value => $K = \frac{1}{t_c}(\frac{E}{1-E})$ is a fixed value
  - $W = KT_0$(T_0 is a function of the number of nodes p and the workload w)
- Both the workload W and the overhead h can be expressed as functions of n and s
- So the efficiency can also be expressed as follows
  - $E = \frac{1}{1+\frac{h(s,n)}{W(s)}}$

Isoefficiency Function

- To keep E constant, the workload W(s) should grow proportionally to the overhead h(s,n), which leads to the following conditions
  - $W(s) = \frac{E}{1-E} * h(s,n)$
  - $c = \frac{E}{1-E}$ is a fixed value
  - **Isoefficiency Function**: $F_E(n) = C*h(s,n)$
- If the workload W(s) grows as fast as $F_E(n)$, then the combination of known algorithm structures keeps the efficiency constant
- In order to get the same efficiency, it is enough to make W(s) and h(s,n) the same order of magnitude

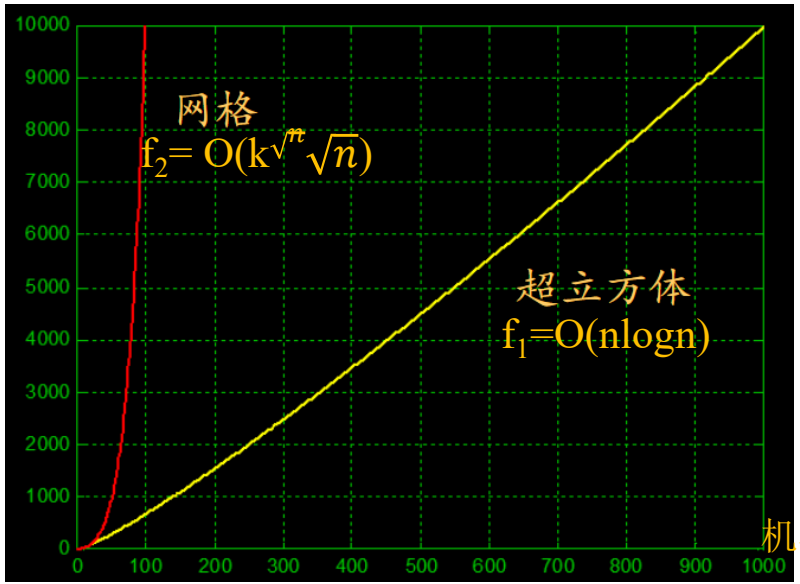
Isoefficiency

Example

- 保持恒等效率的情况下，在n台处理机网格和超立方体计算机上分别计算1维s点的FFT
- 工作负载W(s)=O(slogs)
- 已知：
  - 超立方体计算机上：$h_1(s, n) = O(n\log n + s\log n)$
  - 网格计算机上：$h_2(s, n) = O(n\log n + s\sqrt{n})$
- 问哪一种扩展性好?

Isoefficiency

- 对超立方体计算机
  - 有O(slogs) = O(nlogn) => s=n
  - 或O(slogs) = O(slogn) => s=n
  - => f_1=W(s)= O(nlogn)
- 对网格计算机
  - 有O(slogs) = O(nlogn) => s=n
  - 或O(slog$_k$s) = O(s$\sqrt{n}$) => s=k$^{\sqrt{n}}$
  - => f_2=W(s)= O(slogs) = O(k$^{\sqrt{n}}\sqrt{n}$)
- 为了得到恒等效率
  - 网格计算机的负载必须以指数增长，
  - 超立方体的负载的增长不超过多项式增长速度
- 结论:超立方体具有更好的可扩展性



对于相同的效率E,设k=2,机器规模-问题规模曲线

Conclusion

- Performance analysis
  - Speedup
    - ✓ linear speedup
    - ✓ super-linear speedup
    - ✓ sub-linear speedup
  - Efficiency
  - Scalability
    - ✓ Asymptotic analysis
      - Strong scalability
        - Amdahl's law
      - Weak scalability
        - Gustafson's law
      - Isoefficiency