Modelagem Analítica de Programas Paralelos

Tempo de Execução Seqüencial e Paralelo

Aula 17
Alessandro L. Koerich

Programação Paralela
Pontifícia Universidade Católica do Paraná (PUCPR)
Ciência da Computação – 6º Período

Programa do PA

Aula Anterior

- Métodos para Restringir Perdas (*Overhead*) Causadas por Interações.
- Modelos de Algoritmos Paralelos

Plano de Aula

- Fontes de Perdas
- Métricas de Performance
- Efeito da Granularidade
- Escalabilidade de Sistemas Paralelos

Introdução

- Um algoritmo seqüencial é geralmente avaliado em termos de seu *tempo de execução*, expresso como uma função do tamanho da sua entrada.
- O tempo de execução de um algoritmo paralelo não depende somente do tamanho da entrada mas também do número de elementos de processamento usados, suas computações relativas e velocidades de comunicação entre processos.
- Portanto, um algoritmo paralelo não pode ser avaliado sem considerar uma arquitetura paralela sem alguma perda de precisão.

Introdução

- Um sistema paralelo é a combinação de um algoritmo com a arquitetura paralela sobre a qual ele é implementado.
- Na aula de hoje vamos estudar várias métricas para avaliação da performance de sistemas paralelos.

Fontes de Perdas

- Usando duas vezes mais recursos de hardware, seria razoável esperar que um programa seja executado duas vezes mais rápido
- Entretanto, em programas paralelos típicos, isto raramente acontece devido a uma variedade de perdas (*overheads*) associadas com o paralelismo
- Uma quantificação precisa destas perdas é crítico para compreender a performance de programas paralelos.
- Ver figura 5.1

Fontes de Perdas

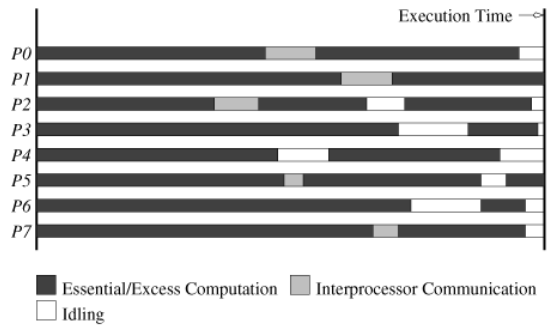


Figure 5.1 The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Fontes de Perdas:
Interação entre Processos

- Qualquer sistema paralelo não trivial necessita que seus elementos de processamento interajam e se comuniquem (e.g., troca de resultados intermediários).
- O tempo despendido para realizar a comunicação de dados entre elementos de processamento é geralmente a fonte mais significativa de perdas em processamento paralelo.

Fontes de Perdas:
Ociosidade

- Elementos de processamento em um sistema paralelo podem se tornar ociosos devido a muitos razões, como por exemplo:
  - desbalanceamento de carga;
  - Sincronização;
  - presença de componentes seriais em um programa;
- Em muitas aplicações paralelas (e.g. geração de tarefas dinâmica), é impossível (ou pelo menos difícil) predizer o tamanho das subtarefas atribuídas a vários elementos de processamento.

Fontes de Perdas:
Ociosidade

- Em alguns programas paralelos os elementos de processamento devem sincronizar em determinados pontos da execução.
- Partes de um algoritmo podem ser "*não paralelizáveis*", permitindo somente que um único elemento de processamento trabalhe sobre ele.

Fontes de Perdas:
Computação Excessiva

- O mais rápido algoritmo seqüencial conhecido para um dado problema pode ser difícil ou mesmo impossível de paralelizar.
- **Solução**: utilizar um algoritmo paralelo baseado em um algoritmo seqüencial facilmente paralelizável, porém, cuja performance é reconhecidamente inferior.
- A diferença em computação realizada pelo programa paralelo e o melhor programa serial é a *perda por excesso de computação* causada pelo programa paralelo.

Fontes de Perdas

Como diferentes algoritmos paralelos usado para resolver os mesmos problemas causam perdas diversificadas, é importante quantificar estas perdas em vista de estabelecer uma figura de mérito para cada algoritmo.

Métricas de Performance para Sistemas Paralelos

- É importante estudar a performance de algoritmos paralelos no intuito de determinar o melhor algoritmo, avaliando plataformas de hardware e examinando os benefícios do paralelismo.
- Diversas métricas tem sido utilizadas baseadas no resultado desejado da análise de performance.

Métricas de Performance:
Tempo de Execução

- O *tempo de execução serial* (T_S) de um programa é o tempo decorrido entre o início e o final de sua execução em um computador seqüencial.
- O *tempo de execução paralelo* (T_P) é o tempo transcorrido a partir do momento que uma computação paralela começa até o momento em que o último elemento de processamento finaliza a execução.

Métricas de Performance:
Perdas Paralelas Totais

- As perdas causadas por um programa paralelo são encapsuladas dentro de uma única expressão: *função de perdas*
- Definimos a *função de perdas* ou *perda total de um sistema paralelo* como:
  "*tempo total gasto coletivamente* por todos os elementos de processamento sobre aquele necessário para que o mais rápido algoritmo seqüencial conhecido resolva o mesmo problema sobre um único elemento de processamento".

Métricas de Performance:
Perdas Paralelas Totais

- O tempo total gasto resolvendo um problema somado sobre todos os elementos de processamento é pT_P.
- T_S unidades desse tempo são gastas realizando trabalho útil e o resto são perdas.
- Portanto, a *função de perdas* (T_o) é dada por:

$$T_o = pT_P - T_S$$

Métricas de Performance:
Aceleração (*Speedup*)

- Quando avaliamos um sistema paralelo estamos geralmente interessados em conhecer qual o ganho de performance obtido pela paralelização de uma dada aplicação sobre uma implementação seqüencial.
- ***Speedup*** (S): é uma medida que captura o benefício relativo de resolver um problema em paralelo.
- *Speedup* é definido como a razão entre o tempo gasto para resolver o problema em um único elemento de processamento e tempo necessário para resolver o mesmo problema em um computador paralelo com p elementos de processamento idênticos.

Métricas de Performance:
Aceleração (*Speedup*)

- Comparamos a performance de um algoritmo paralelo para resolver um problema com aquela obtida pelo melhor algoritmo seqüencial que resolve o mesmo problema.
- Teoricamente, o *speedup* nunca pode exceder o número de elementos de processamento p.

Métricas de Performance:
Aceleração (*Speedup*)

- Na prática, um *speedup* maior do que p pode ser observado algumas vezes (um fenômeno conhecido como ***superlinear speedup***).
- Isto geralmente acontece quando o trabalho realizado por um algoritmo serial é maior que sua formulação paralela ou devido a características do hardware que colocam a implementação serial em desvantagem. Exemplo: grandes dados no cache

Métricas de Performance:
Aceleração (*Speedup*)

Example 5.1 Adding n numbers using n processing elements

Consider the problem of adding n numbers by using n processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end of the computation, one of the processing elements stores the sum of all the numbers. Assuming that n is a power of two, we can perform this operation in $\log n$ steps by propagating partial sums up a logical binary tree of processing elements. Figure 5.2 illustrates the procedure for $n = 16$. The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of the numbers with consecutive labels from i to j is denoted by Σ_i^j.

Each step shown in Figure 5.2 consists of one addition and the communication of a single word. The addition can be performed in some constant time, say t_c, and the communication of a single word can be performed in time $t_s + t_w$. Therefore, the addition and communication operations take a constant amount of time. Thus,

$$T_P = \Theta(\log n). \quad (5.2)$$

Since the problem can be solved in $\Theta(n)$ time on a single processing element, its speedup is

$$S = \Theta\left(\frac{n}{\log n}\right). \quad (5.3)$$

■

Métricas de Performance:
Aceleração (*Speedup*)

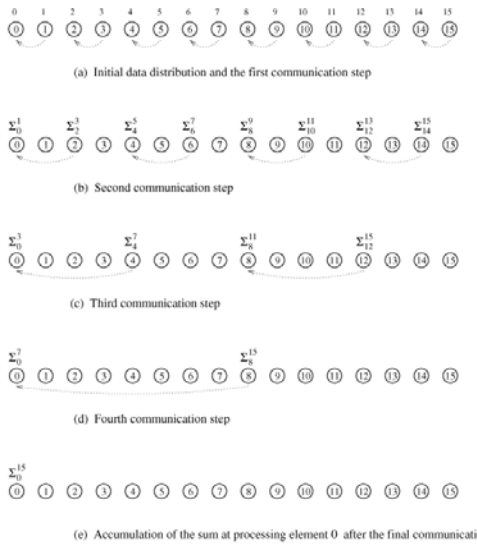


Figure 5.2 Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Métricas de Performance:
Aceleração (*Speedup*)

Example 5.2 Computing speedups of parallel programs

Consider the example of parallelizing bubble sort (Section 9.3.1). Assume that a serial version of bubble sort of 10^5 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. ■

Métricas de Performance:
Aceleração (*Speedup*)

Example 5.3 Superlinearity effects from caches

Consider the execution of a parallel program on a two-processor parallel system. The program attempts to solve a problem instance of size W. With this size and available cache of 64 KB on one processor, the program has a cache hit rate of 80%. Assuming the latency to cache of 2 ns and latency to DRAM of 100 ns, the effective memory access time is $2 \times 0.8 + 100 \times 0.2$, or 21.6 ns. If the computation is memory bound and performs one FLOP/memory access, this corresponds to a processing rate of 46.3 MFLOPS. Now consider a situation when each of the two processors is effectively executing half of the problem instance (i.e., size $W/2$). At this problem size, the cache hit ratio is expected to be higher, since the effective problem size is smaller. Let us assume that the cache hit ratio is 90%, 8% of the remaining data comes from local DRAM, and the other 2% comes from the remote DRAM (communication overhead). Assuming that remote data access takes 400 ns, this corresponds to an overall access time of $2 \times 0.9 + 100 \times 0.08 + 400 \times 0.02$, or 17.8 ns. The corresponding execution rate at each processor is therefore 56.18, for a total execution rate of 112.36 MFLOPS. The speedup in this case is given by the increase in speed over serial formulation, i.e., 112.36/46.3 or 2.43! Here, because of increased cache hit ratio resulting from lower problem size per processor, we notice superlinear speedup. ■

Métricas de Performance:
Aceleração (*Speedup*)

Example 5.4 Superlinearity effects due to exploratory decomposition

Consider an algorithm for exploring leaf nodes of an unstructured tree. Each leaf has a label associated with it and the objective is to find a node with a specified label, in this case 'S'. Such computations are often used to solve combinatorial problems, where the label 'S' could imply the solution to the problem (Section 11.6). In Figure 5.3, we illustrate such a tree. The solution node is the rightmost leaf in the tree. A serial formulation of this problem based on depth-first tree traversal explores the entire tree, i.e., all 14 nodes. If it takes time t_c to visit a node, the time for this traversal is $14t_c$. Now consider a parallel formulation in which the left subtree is explored by processing element 0 and the right subtree by processing element 1. If both processing elements explore the tree at the same speed, the parallel formulation explores only the shaded nodes before the solution is found. Notice that the total work done by the parallel algorithm is only nine node expansions, i.e., $9t_c$. The corresponding parallel time, assuming the root node expansion is serial, is $5t_c$ (one root node expansion, followed by four node expansions by each processing element). The speedup of this two-processor execution is therefore $14t_c/5t_c$, or 2.8!

The cause for this superlinearity is that the work performed by parallel and serial algorithms is different. Indeed, if the two-processor algorithm was implemented as two processes on the same processing element, the algorithmic superlinearity would disappear for this problem instance. Note that when exploratory decomposition is used, the relative amount of work performed by serial and parallel algorithms is dependent upon the location of the solution, and it is often not possible to find a serial algorithm that is optimal for all instances. Such effects are further analyzed in greater detail in Chapter 11. ■

Métricas de Performance:
Aceleração (*Speedup*)

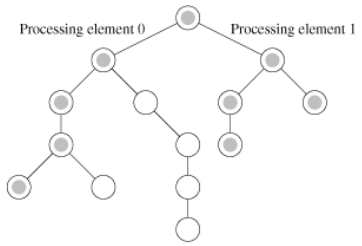


Figure 5.3 Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.

Métricas de Performance:
Eficiência

- Somente um sistema paralelo ideal contendo *p* elementos de processamento pode fornecer um *speedup* igual a *p*.
- Na prática, este comportamento ideal não é atingido porque durante a execução de um algoritmo paralelo, os elementos de processamento não podem dedicar 100% de seu tempo para a computação dos algoritmos

Métricas de Performance:
Eficiência

- ***Eficiência* (*S*)**: é uma medida da fração de tempo para o qual um elemento de processamento é empregado de forma proveitosa. É definida como a razão do speedup e do número de elementos de processamento (Ideal: $S=p$, $E=1$)

.

$$E = \frac{S}{P}$$

Métricas de Performance:
Eficiência

Example 5.5 Efficiency of adding n numbers on n processing elements
From Equation 5.3 and the preceding definition, the efficiency of the algorithm for adding n numbers on n processing elements is

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n} = \Theta\left(\frac{1}{\log n}\right)$$

■

Métricas de Performance:
Custo

- Definimos o ***custo*** de resolver um problema em um sistema paralelo como o produto do tempo de execução paralelo e o número de elementos de processamento utilizados.
- O custo de resolver um problema em um único elemento paralelo é o tempo de execução do mais rápido algoritmo seqüencial conhecido.

Métricas de Performance:
Custo

- Um sistema paralelo é dito ser de custo ótimo se o custo para resolver um problema em um computador paralelo tiver o mesmo crescimento assintótico (em termos de Θ) que uma função do tamanho da entrada do mais rápido algoritmo seqüencial conhecido em um único processador.
- Custo é algumas vezes chamado de ***trabalho*** ou ***produto tempo–processador***, e um sistema de custo ótimo é também conhecido como um sistema pT_P–ótimo.

Métricas de Performance:
Custo

Example 5.7 Cost of adding n numbers on n processing elements
The algorithm given in Example 5.1 for adding n numbers on n processing elements has a processor-time product of $\Theta(n \log n)$. Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal. ■

Métricas de Performance: Custo

Example 5.8 Performance of non-cost optimal algorithms

Consider a sorting algorithm that uses n processing elements to sort the list in time $(\log n)^2$. Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $n/\log n$ and $1/\log n$, respectively. The pT_P product of this algorithm is $n(\log n)^2$. Therefore, this algorithm is not cost optimal but only by a factor of $\log n$. Let us consider a realistic scenario in which the number of processing elements p is much less than n. An assignment of these n tasks to $p < n$ processing elements gives us a parallel time less than $n(\log n)^2/p$. This follows from the fact that if n processing elements take time $(\log n)^2$, then one processing element would take time $n(\log n)^2$; and p processing elements would take time $n(\log n)^2/p$. The corresponding speedup of this formulation is $p/\log n$. Consider the problem of sorting 1024 numbers ($n = 1024$, $\log n = 10$) on 32 processing elements. The speedup expected is only $p/\log n$ or 3.2. This number gets worse as n increases. For $n = 10^6$, $\log n = 20$ and the speedup is only 1.6. Clearly, there is a significant cost associated with not being cost-optimal even by a very small factor (note that a factor of $\log p$ is smaller than even $\sqrt{p}$). This emphasizes the practical importance of cost-optimality. ■

Efeito da Granularidade na Performance

- Na prática, atribuímos grandes pedaços dos dados de entrada aos elementos de processamento.
- Isto corresponde a aumentar a granularidade da computação nos elementos de processamento.
- Usando menos do que o número máximo possível de elementos de processamento para executar um algoritmo paralelo é chamado ***scaling down*** um sistema paralelo em termos do número de elementos de processamento.

Efeito da Granularidade na Performance

- Uma maneira simples de *scale down* um sistema paralelo é projetar um algoritmo paralelo para um elemento de entrada por elemento de processamento, e então usar menos elementos de processamento para simular um grande número de elementos de processamento.
- Se houverem n entradas e somente p elementos de processamento ($p < n$), podemos usar o algoritmo paralelo projetado para n elementos de processamento assumindo n elementos de processamento virtuais e tendo cada um dos p elementos de processamento físicos simulam n/p elementos de processamento virtuais.

Efeito da Granularidade na Performance

- A medida que o número de elementos de processamento decresce por um fator n/p, a computação em cada elemento de processamento aumenta por um fator n/p porque cada elemento de processamento agora realiza o trabalho de n/p elementos de processamento.
- Se elementos de processamento virtual são mapeados apropriadamente em elementos de processamento físicos, o tempo de comunicação total não cresce por mais que uma fator n/p.

Efeito da Granularidade na Performance

- O tempo de execução paralelo total aumenta, no máximo, por um fator n/p, e o produto *processador–tempo* não aumenta. Portanto, se uma sistema paralelo com n elementos de processamento é custo–ótimo, usando p elementos de processamento (onde $p < n$) para simular n elementos de processamento preserva custo–otimalidade.

Efeito da Granularidade na Performance

- Uma desvantagem deste método simples de aumentar a granularidade computacional é que se um sistema paralelo não é custo–ótimo para iniciar com, ele pode ainda não ser custo–ótimo após a granularidade da computação aumenta.
- Estes exemplos simples demonstram que a maneira na qual a computação é mapeada em elementos de processamento podem determinar se um sistema paralelo tem custo ótimo.
- Entretanto, note que não podemos fazer todos os sistemas de custo não ótimo em custo ótimo escalando para baixo o número de elementos de processamento.

Efeito da Granularidade na Performance

Example 5.9 Adding n numbers on p processing elements

Consider the problem of adding n numbers on p processing elements such that $p < n$ and both n and p are powers of 2. We use the same algorithm as in Example 5.1 and simulate n processing elements on p processing elements. The steps leading to the solution are shown in Figure 5.5 for $n = 16$ and $p = 4$. Virtual processing element i is simulated by the physical processing element labeled $i \bmod p$; the numbers to be added are distributed similarly. The first $\log p$ of the $\log n$ steps of the original algorithm are simulated in $(n/p) \log p$ steps on p processing elements. In the remaining steps, no communication is required because the processing elements that communicate in the original algorithm are simulated by the same processing element; hence, the remaining numbers are added locally. The algorithm takes $\Theta((n/p) \log p)$ time in the steps that require communication, after which a single processing element is left with n/p numbers to add, taking time $\Theta(n/p)$. Thus, the overall parallel execution time of this parallel system is $\Theta((n/p) \log p)$. Consequently, its cost is $\Theta(n \log p)$, which is asymptotically higher than the $\Theta(n)$ cost of adding n numbers sequentially. Therefore, the parallel system is not cost-optimal. ■

Efeito da Granularidade na Performance

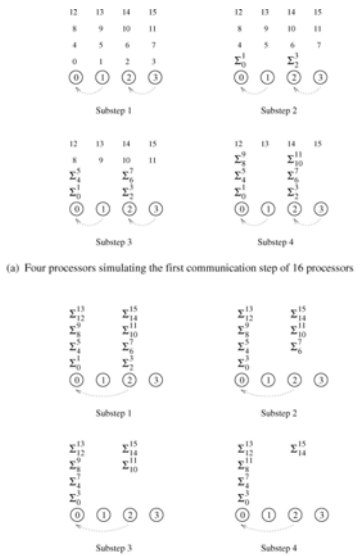
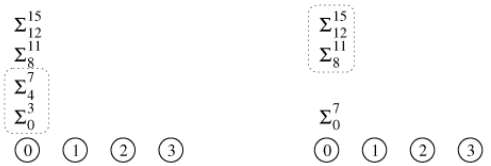
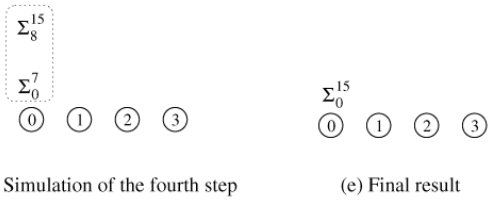


Figure 5.5 Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (first two steps). Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Efeito da Granularidade na Performance



(c) Simulation of the third step in two substeps



(d) Simulation of the fourth step (e) Final result

Figure 5.5 (continued) Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).

Escalabilidade de Sistemas Paralelos

- Freqüentemente, programas são projetados e testados em para pequenos problemas sobre poucos elementos de processamento. Entretanto, os problemas reais que estes programas são destinados a resolver são muito maiores e as máquinas contém um número maior de elementos de processamento.
- Apesar do desenvolvimento do código ser simplificado pela utilização de versões menores (*scaled–down*) da máquina e dos problemas, sua performance e correção (dos programas) é muito mais difícil de estabelecer baseando–se em sistemas *scaled–down*.
- Investigaremos técnicas para avaliar a escalabilidade de programas paralelos usando ferramentas analíticas.

Escalabilidade de Sistemas Paralelos

Example 5.11 Why is performance extrapolation so difficult?

Consider three parallel algorithms for computing an n-point Fast Fourier Transform (FFT) on 64 processing elements. Figure 5.7 illustrates speedup as the value of n is increased to 18 K. Keeping the number of processing elements constant, at smaller values of n, one would infer from observed speedups that binary exchange and 3-D transpose algorithms are the best. However, as the problem is scaled up to 18 K points or more, it is evident from Figure 5.7 that the 2-D transpose algorithm yields best speedup. (These algorithms are discussed in greater detail in Chapter 13.) ■

Escalabilidade de Sistemas Paralelos

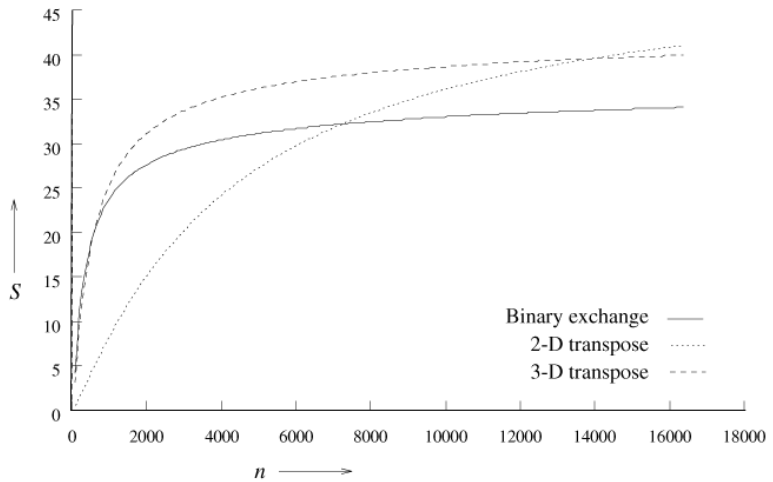


Figure 5.7 A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$ (see Chapter 13 for details).

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

- A eficiência de um programa paralelo pode ser escrita como:

$$E = \frac{S}{P} = \frac{T_S}{pT_P}$$

- Usando a expressão para perdas paralelas, podemos re–escrever esta expressão:

$$E = \frac{1}{1 + \frac{T_o}{T_S}}$$

- A função de perdas total (T_o) é uma função crescente de p, pois todo programa deve conter algum componente serial. Se este componente serial do programa leva tempo t_{serial}, então durante este tempo todos os outros elementos de processamento devem estar ociosos.

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

- Isto corresponde a função de perda total de $(p–1)\times t_{serial}$. Portanto, *To* cresce pelo menos linearmente com p.

- Em adição, devido a comunicação, ociosidade, e excesso de computação, esta função pode crescer superlinearmente com o número de elementos de processamento.

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

- A partir da equação anterior:
  - Para um dado tamanho de problema, (i.e. o valor de T_S permanece constante), a medida que aumentamos o número de elementos de processamento, T_o aumenta.
  - Em tal cenário, é claro que a eficiência total do programa paralelo cai.
  - Esta característica de diminuição da eficiência com um aumento do número de elementos de processamento para um tamanho de problema dado é comum para todos programas paralelos.

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

Example 5.12 Speedup and efficiency as functions of the number of processing elements

Consider the problem of adding n numbers on p processing elements. We use the same algorithm as in Example 5.10. However, to illustrate actual speedups, we work with constants here instead of asymptotics. Assuming unit time for adding two numbers, the first phase (local summations) of the algorithm takes roughly n/p time. The second phase involves $\log p$ steps with a communication and an addition at each step. If a single communication takes unit time as well, the time for this phase is $2 \log p$. Therefore, we can derive parallel time, speedup, and efficiency as:

$$T_P = \frac{n}{p} + 2 \log p \quad (5.7)$$

$$S = \frac{n}{\frac{n}{p} + 2 \log p} \quad (5.8)$$

$$E = \frac{1}{1 + \frac{2p \log p}{n}} \quad (5.9)$$

These expressions can be used to calculate the speedup and efficiency for any pair of n and p. Figure 5.8 shows the S versus p curves for a few different values of n and p. Table 5.1 shows the corresponding efficiencies.

Figure 5.8 and Table 5.1 illustrate that the speedup tends to saturate and efficiency drops as a consequence of ***Amdahl's law*** (Problem 5.1). Furthermore, a larger instance of the same problem yields higher speedup and efficiency for the same number of processing elements, although both speedup and efficiency continue to drop with increasing p. ■

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

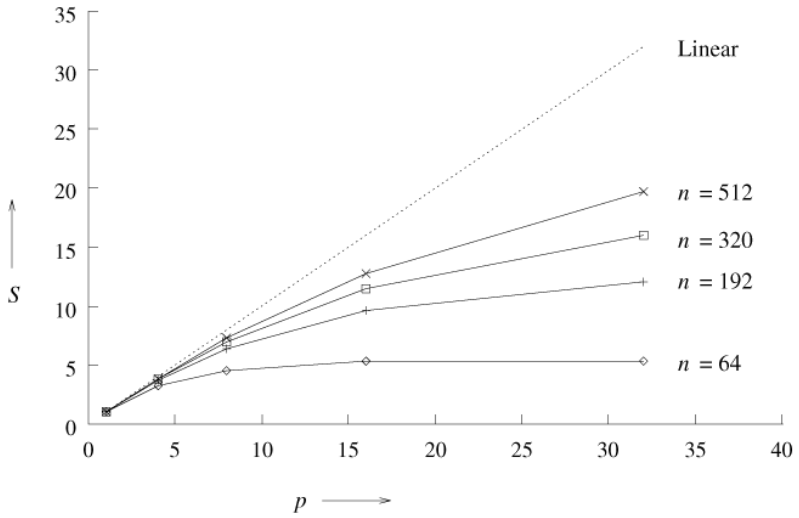


Figure 5.8 Speedup versus the number of processing elements for adding a list of numbers.

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

Table 5.1 Efficiency as a function of n and p for adding n numbers on p processing elements.

n	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
64	1.0	***0.80***	0.57	0.33	0.17
192	1.0	0.92	***0.80***	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	***0.80***	0.62

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

- Efeito de aumentar o tamanho do problema mantendo o número de elementos de processamento constante.
- A habilidade de manter a eficiência em um valor fixo, pelo aumento simultâneo do número de elementos de processamento e o tamanho do problema é apresentado por muitos sistemas paralelos.
- Chamamos estes sistemas de sistemas paralelo escaláveis.

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

- A escalabilidade de um sistema paralelo, é uma medida de sua capacidade em aumentar o *speedup* em proporção ao número de elementos de processamento.
- Reflete a capacidade de um sistema paralelo em utilizar recursos paralelos crescentes com eficiência.

Escalabilidade de Sistemas Paralelos:
Características de Escalamento

Example 5.13 Scalability of adding n numbers

For the cost-optimal addition of n numbers on p processing elements $n = \Omega(p \log p)$. As shown in Table 5.1, the efficiency is 0.80 for $n = 64$ and $p = 4$. At this point, the relation between n and p is $n = 8p \log p$. If the number of processing elements is increased to eight, then $8p \log p = 192$. Table 5.1 shows that the efficiency is indeed 0.80 with $n = 192$ for eight processing elements. Similarly, for $p = 16$, the efficiency is 0.80 for $n = 8p \log p = 512$. Thus, this parallel system remains cost-optimal at an efficiency of 0.80 if n is increased as $8p \log p$. ■

Escalabilidade de Sistemas Paralelos:
Métrica de Isoeficiência da Escalabilidade

- Resumimos a discussão do ponto anterior:

1. Para um dado tamanho de problema, a medida que aumentamos o # de processadores, a eficiência global do sistema paralelo cai. Fenômeno comum.

2. Em muitos casos, a eficiência do um sistema paralelo aumenta se o tamanho do problema é aumentado, enquanto o # de processadores é mantido constante.

- Figura 5.9

Escalabilidade de Sistemas Paralelos:
Métrica de Isoeficiência da Escalabilidade

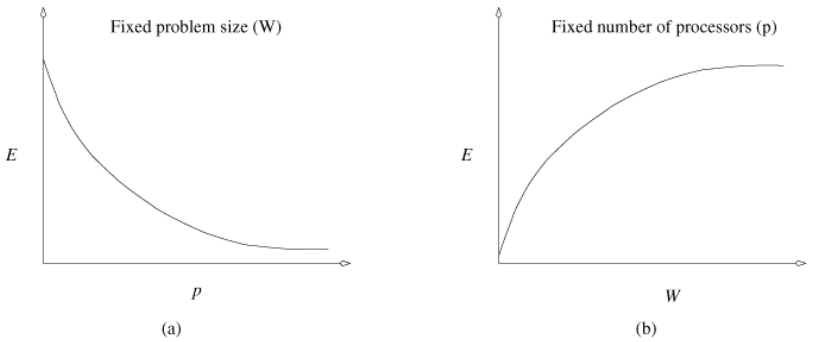


Figure 5.9 Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.

Escalabilidade de Sistemas Paralelos:
Métrica de Isoeficiência da Escalabilidade

Tamanho de Problema

- Uma maneira simples de expressar o tamanho de um problema é como um parâmetro do tamanho da entrada; por exemplo, n no caso de uma operação matricial envolvendo $n \times n$ matrizes.

- Uma desvantagem desta definição é que a interpretação do tamanho do problema muda de acordo com o problema

Tamanho de Problema

- Por exemplo, dobrando o tamanho da entrada resulta um aumento de 8 vezes no tempo de execução para a multiplicação de matrizes, e 4 vezes para a adição de matrizes. (algoritmo $\Theta(n^3)$ convencional).
- Definição consistente: expressar o tamanho de um problema em termos do número total de operações básicas necessárias para resolver um problema (considerando sempre um único processador e o melhor algoritmo seqüencial)

Função Isoeficiência

- Tempo de execução paralelo pode ser expresso como uma função do tamanho do problema, função de perdas e número de processadores:

.

$$T_P = \frac{W + T_o(W, p)}{p}$$

$$S = \frac{W}{T_P} = \frac{Wp}{W + T_o(W, p)}$$

$$E = \frac{S}{P} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + \frac{T_o(W, p)}{W}}$$

Função Isoeficiência

- Para sistemas paralelo escaláveis, a eficiência pode ser mantida em um valor fixo [0,1] se a razão T_o/W for mantida em um valor constante. Para um valor de eficiência desejado (E):

.

$$E = \frac{1}{1 + \frac{T_o(W, p)}{W}}$$

$$\frac{T_o(W, p)}{W} = \frac{1 - E}{E}$$

$$W = \frac{E}{1 - E} T_o(W, p)$$

Função Isoeficiência

- Em uma única expressão, a função isoeficiencia captura as características de um algoritmo paralelo, bem como a arquitetura paralela onde na qual ele é implementado.
- Após realizar a análise de isoeficiencia, podemos testar a performance de um algoritmo paralelo sobre poucos processadores e então predizer sua performance sobre um número grande de processadores.

Escalabilidade de Sistemas Paralelos:
Métrica de Isoeficiência da Escalabilidade

Função Isoeficiência

- Entretanto, a utilidade da análise de isoeficiência não está limitada em predizer o impacto na performance de um número crescente de elementos de processamento.
- A análise de isoeficiencia pode também ser usada para estudar o comportamento de um sistema paralelo em relação a mudanças nos parâmetros de hardware como velocidade dos processadores e canais de comunicação.

Escalabilidade de Sistemas Paralelos:
Métrica de Isoeficiência da Escalabilidade

Função Isoeficiência

Example 5.14 Isoefficiency function of adding numbers

The overhead function for the problem of adding n numbers on p processing elements is approximately $2p \log p$, as given by Equations 5.9 and 5.1. Substituting T_o by $2p \log p$ in Equation 5.14, we get

$$W = K2p \log p. \quad (5.15)$$

Thus, the asymptotic isoefficiency function for this parallel system is $\Theta(p \log p)$. This means that, if the number of processing elements is increased from p to p', the problem size (in this case, n) must be increased by a factor of $(p' \log p')/(p \log p)$ to get the same efficiency as on p processing elements. In other words, increasing the number of processing elements by a factor of p'/p requires that n be increased by a factor of $(p' \log p')/(p \log p)$ to increase the speedup by a factor of p'/p. ■

Escalabilidade de Sistemas Paralelos:
Métrica de Isoeficiência da Escalabilidade

Função Isoeficiência

Example 5.15 Isoefficiency function of a parallel system with a complex overhead function

Consider a parallel system for which $T_o = p^{3/2} + p^{3/4}W^{3/4}$. Using only the first term of T_o in Equation 5.14, we get

$$W = Kp^{3/2}. \quad (5.16)$$

Using only the second term, Equation 5.14 yields the following relation between W and p:

$$\begin{aligned} W &= Kp^{3/4}W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4p^3 \end{aligned} \quad (5.17)$$

To ensure that the efficiency does not decrease as the number of processing elements increases, the first and second terms of the overhead function require the problem size to grow as $\Theta(p^{3/2})$ and $\Theta(p^3)$, respectively. The asymptotically higher of the two rates, $\Theta(p^3)$, gives the overall asymptotic isoefficiency function of this parallel system, since it subsumes the rate dictated by the other term. The reader may indeed verify that if the problem size is increased at this rate, the efficiency is $\Theta(1)$ and that any rate lower than this causes the efficiency to fall with increasing p. ■

Escalabilidade de Sistemas Paralelos:
Custo–Otimalidade e a Função de Isoeficiência

- Um sistema paralelo é custo–ótimo se o produto do # processadores e o tempo de execução paralela é proporcional ao tempo de execução do mais rápido algoritmo seqüencial em um único processador. Em outras palavras, um sistema paralelo é custo–ótimo sss:

$$\begin{aligned} pT_P &= \Theta(W) \\ W + T_o(W, p) &= \Theta(W) \\ T_o(W, p) &= O(W) \\ W &= \Omega(T_o(W, p)) \end{aligned}$$

.

Escalabilidade de Sistemas Paralelos:
Custo–Otimalidade e a Função de Isoeficiência

Example 5.16 Relationship between cost-optimality and isoefficiency

Consider the cost-optimal solution to the problem of adding n numbers on p processing elements, presented in Example 5.10. For this parallel system, $W \approx n$, and $T_o = \Theta(p \log p)$. From Equation 5.14, its isoefficiency function is $\Theta(p \log p)$; that is, the problem size must increase as $\Theta(p \log p)$ to maintain a constant efficiency. In Example 5.10 we also derived the condition for cost-optimality as $W = \Omega(p \log p)$.

■

Tempo Mínimo de Execução

- Estamos geralmente interessados em saber quão rápido um problema pode ser resolvido, ou qual o mínimo tempo de execução possível para um algoritmo paralelo, sendo que o # de processadores não é uma restrição.
- A medida que aumentamos o # de processadores para um dado tamanho de problema, ou o tempo de execução paralelo continua a decair e aproxima assintóticamente um valor mínimo, ou ele começa a subir após atingir um valor mínimo.

Tempo Mínimo de Execução

- Podemos determinar o mínimo tempo de execução paralela T_P^{min} para um dado W através da diferenciação da expressão para T_P em relação a p e igualando a derivada a zero.
- O número de processadores cujo T_P é mínimo é determinado pela seguinte equação:

$$\frac{d}{dp} T_p = 0$$

Tempo Mínimo de Execução

Example 5.18 Minimum execution time for adding n numbers

Under the assumptions of Example 5.12, the parallel run time for the problem of adding n numbers on p processing elements can be approximated by

$$T_P = \frac{n}{p} + 2 \log p. \tag{5.22}$$

Equating the derivative with respect to p of the right-hand side of Equation 5.22 to zero we get the solutions for p as follows:

$$\begin{aligned} -\frac{n}{p^2} + \frac{2}{p} &= 0 \\ -n + 2p &= 0 \\ p &= \frac{n}{2} \end{aligned} \tag{5.23}$$

Substituting $p = n/2$ in Equation 5.22, we get

$$T_P^{min} = 2 \log n. \tag{5.24}$$

■

Tempo Mínimo de Execução

- Derivamos, agora, um resultado importante que fornece um limite inferior sobre o tempo de execução paralelo se o problema é resolvido com custo ótimo.
- Sendo $T_P^{cost\_opt}$ o tempo mínimo no qual um problema pode ser resolvido por um sistema paralelo custo ótimo, se a função de isoeficiência de uma sistema paralelo é $\Theta(f(p))$, então um problema de tamanho W pode ser resolvido com custo ótimo somente se $W = \Omega(f(p))$.

Tempo Mínimo de Execução

- Ou seja, dado um problema de tamanho W, uma solução de custo ótimo necessita que $p = O(f^{-1}(W))$.
- Como o tempo de execução paralelo é $\Theta(W/p)$ para um sistema para de custo ótimo, o limite inferior sobre o tempo de execução paralelo para resolver um problema de tamanho W com custo ótimo é:

.

$$T_p^{\cos t\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right)$$

Tempo Mínimo de Execução

Example 5.19 Minimum cost-optimal execution time for adding n numbers As derived in Example 5.14, the isoefficiency function $f(p)$ of this parallel system is $\Theta(p \log p)$. If $W = n = f(p) = p \log p$, then $\log n = \log p + \log \log p$. Ignoring the double logarithmic term, $\log n \approx \log p$. If $n = f(p) = p \log p$, then $p = f^{-1}(n) = n/\log p \approx n/\log n$. Hence, $f^{-1}(W) = \Theta(n/\log n)$. As a consequence of the relation between cost-optimality and the isoefficiency function, the maximum number of processing elements that can be used to solve this problem cost-optimally is $\Theta(n/\log n)$. Using $p = n/\log n$ in Equation 5.2, we get

$$\begin{aligned} T_P^{cost\ opt} &= \log n + \log\left(\frac{n}{\log n}\right) \\ &= 2\log n - \log\log n. \end{aligned} \quad (5.26)$$

■

Análise Assintótica

- Vamos aplicar estas métricas para quantificar a performance e escalabilidade de um conjunto de algoritmos paralelos na resolução de um dado problema.
- Problema: ordenação de uma lista de n números.
- Algoritmo seqüêncial mais rápido: $T(n) = O(n \log n)$

Análise Assintótica

- Quatro algoritmos paralelos diferentes: A_1, A_2, A_3 e A_4 para ordenar um lista.
- O tempo de execução paralela juntamente com o número de processadores que eles podem usar é mostrado na Tabela 5.2
- Qual destes algoritmos é o melhor ?????
  - Menor tempo de execução: T_p
  - Eficiência: E

Análise Assintótica

Table 5.2 Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the pT_P product.

Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$

Resumo

- Para usar computadores paralelos eficientemente, grandes problemas devem ser resolvidos a medida que mais elementos de processamento são adicionados.
- Entretanto, quando o tamanho do problema é fixo, o objetivo é alcançar o melhor compromisso entre eficiência e tempo de execução paralelo.
- Na maioria das situações, poder de computação adicional derivado de número crescente de processadores pode ser usado para resolver problemas maiores.

Resumo

- Em algumas situações, entretanto, diferentes maneiras de aumentar o tamanho do problema podem ser aplicadas e uma variedade de restrições podem guiar o escalamento da carga de trabalho em relação ao número de processadores.
- Um importante cenário, é aquele onde queremos fazer o uso mais eficiente de um sistema paralelo, ou seja, queremos que a performance global do sistema paralelo aumente com linearmente com p

Resumo

- Isto somente é possível para sistema paralelos escaláveis, que são aqueles cuja eficiência pode se manter fixa para *p* arbitrariamente grande, pelo simples aumento do tamanho do problema.

- Para estes sistemas é mais adequado utilizar métrica de isoeficiência.