Prestanda och skalbarhet

Grama et al.

Introduction to Parallel Computing
Kapitel 5

Robert Granat

Översikt

- Exekveringstid
- Uppsnabbning
- Effektivitet
- Kostnad
- Kostnadsoptimal algoritm
- Ahmdals lag
- Gustafson-Barsis lag
- Uppskalning av problemstorlek
- Kornighet

- Overhead
- Isoeffektivitetsfunktionen
- Grad av parallellitet
- Minimal exekveringstid
- Scaled speedup
- Generalized scaled speedup
- Size up
- Processor efficiency function

Prestanda och skalbarhet hos parallella system

- **T_s:** seriell exekveringstid
- **T_p:** parallell exekveringstid
- **S:** uppsnabbning = T_s(bästa sekventiella algoritm)/T_p
- **E:** effektivitet = S/p "absolut"
- **Kostnad:** "summan" av alla processorers tid = T_p*p
- **Skalbarhet:** förmåga att uppnå prestanda proportionell mot antalet processorer p
- **Kostnadsoptimalt** system: kostnaden T_p*p är proportionell mot $T_s \Rightarrow E = \Theta(1)$ (sekv tid/parallell kostnad)

Amdahls lag

- Låt β vara den seriella andelen av en parallell algoritm
- $T_p = T_s * \beta + (T_s*(1 - \beta)) / p$
- $S = p/(\beta *p + (1 - \beta))$

Exempel:
$\beta = 0.10, p = 10 \Rightarrow S = 5.26$
$\beta = 0.10, p = 64 \Rightarrow S = 8.77$
$\beta = 0.10, p = inf \Rightarrow S = 10.00$

Ahmdahls lag indikerar att även en mycket liten seriell andel omöjliggör effektiv användning av många processorer!

Gustafson-Barsis lag

- Öka (skala upp) parallella delen av ett program
- Antag att den parallella andelen av ett program är p*(1 - β):

  $\rightarrow T_p = \beta + (1 - \beta)$

- På 1 processor exekveras både den seriella och den parallella delen seriellt:

  $\rightarrow T_s = \beta + (1 - \beta)*p$

- $S = p - (p - 1)* \beta$

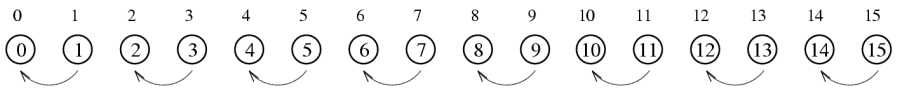
Exempel:
$\beta = 0.10$, p = 10 ==> S = 9.10
$\beta = 0.10$, p = 64 ==> S = 57.60
$\beta = 0.10$, p = inf ==> S = inf

Gustafson-Barsis lag: slutsatser

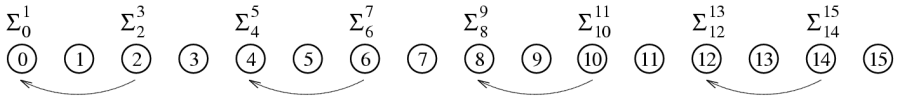
- Gustafson-Barsis lag visar att om problemstorleken skalas då *p* ökar kan mycket hög speedup uppnås även för mycket stora *p*
- Inte alltid möjligt att skala upp enbart den parallella delen (utan att samtidigt skala upp den seriella delen)

Inte självklart hur prestanda ska mätas för skalade problem!

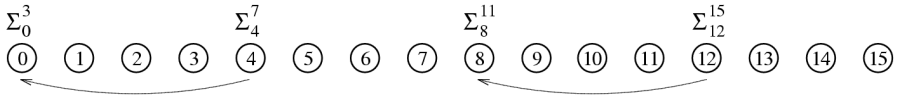
Summera 16 tal på 16-noders hyperkub



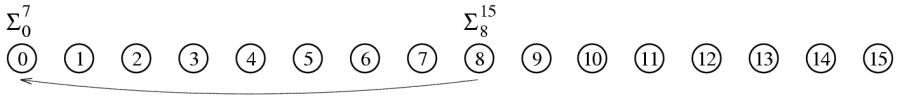
(a) Initial data distribution and the first communication step



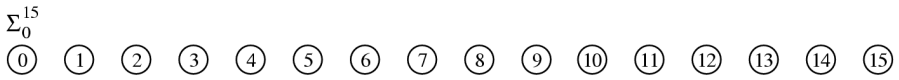
(b) Second communication step



(c) Third communication step



(d) Fourth communication step



(e) Accumulation of the sum at processor 0 after the final communication

Figure 4.1 Computing the sum of 16 numbers on a 16-processor hypercube. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Analys för summering av 16

- Den parallella algoritmen utför 4 steg, vardera med 1 addition och 1 granne-granne kommunikation

$T_p = \log n * (t_a + t_s + t_w) = \Theta(\log n)$

$T_s = n \ t_a = \Theta(n)$

$S = T_s / \ T_p = \Theta(n / \log n)$

$E = S / n = \Theta(1 \ / \log n)$

kostnad = $n * T_p = \Theta(n \log n) > \Theta(n)$,

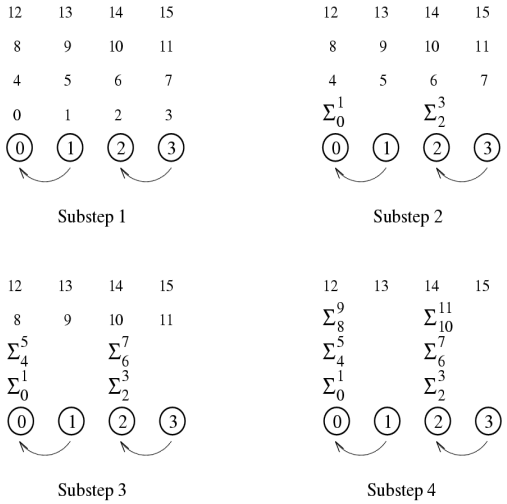
dvs, algoritmen är ej kostnadsoptimal

Kornighet och mappning av data

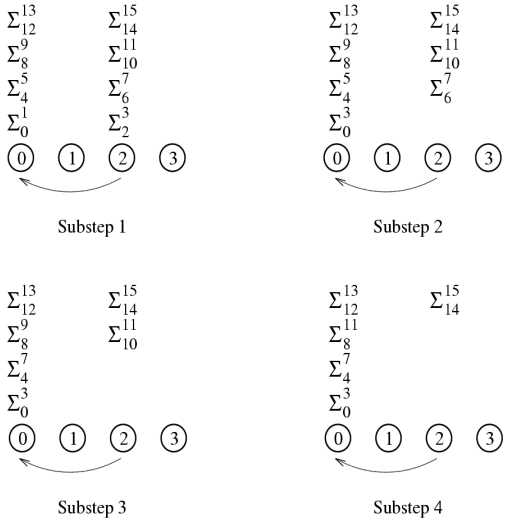
- **Nerskalning** av antal processorer kan förbättra kostnadseffektiviteten
- Nerskalning ger grövre **kornighet**
- Naiv metod (om $p < n$): skapa n virtuella processorer som fördelas på p verkliga processorer
- **Nackdel:** Ej kostnadsoptimal om den ej är det i fallet $p = n$

Effektivare: Ge initialt varje processor större del av arbetet

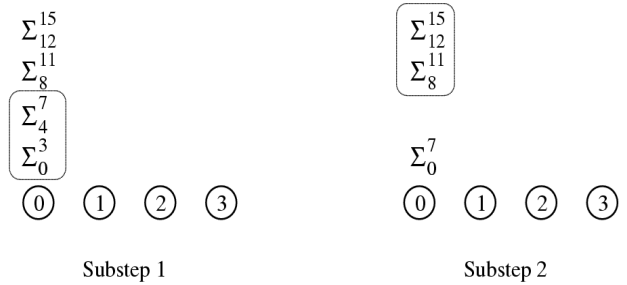
4 processorer simulerar 16



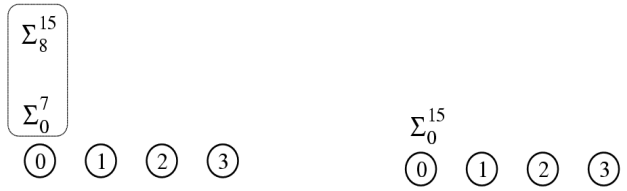
(a) Four processors simulating the first communication step of 16 processors



(b) Four processors simulating the second communication step of 16 processors



(c) Simulation of the third step in two substeps



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

Figure 4.3 (cont.) Four processors simulating 16 processors to compute the sum of 16 numbers (last three steps).

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

$\Longrightarrow p^* T_p = \Theta(n \log p) > \Theta(n)$

Kostnadsoptimalt alternativ

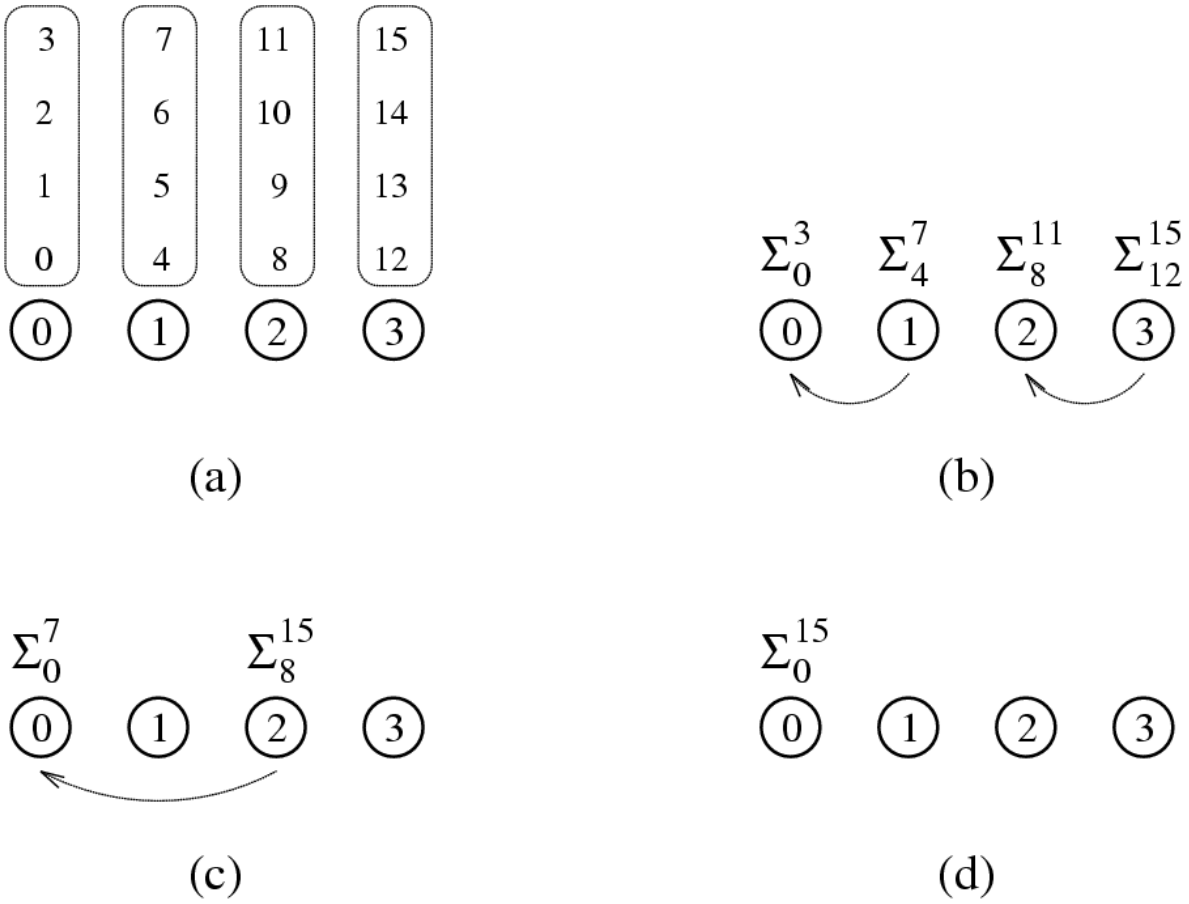


Figure 4.4 A cost-optimal way of computing the sum of 16 numbers on a four-processor hypercube.

Analys för den kostnadsoptimala algoritmen

Varje processor adderar n/p tal lokalt på $\Theta(n / p)$ tid

p delresultat summeras på $\Theta(\log p)$ tid

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

kostnad = $p * T_p = \Theta(n + p \log p)$,
dvs, algoritmen är kostnadsoptimal om $n = \Omega(p \log p)$

Speedup & effektivitet: beror av n och p

Speedup och effektivitet för den kostnadsoptimala algoritmen (antag att en tidsenhet = $t_a = t_s + t_w$)

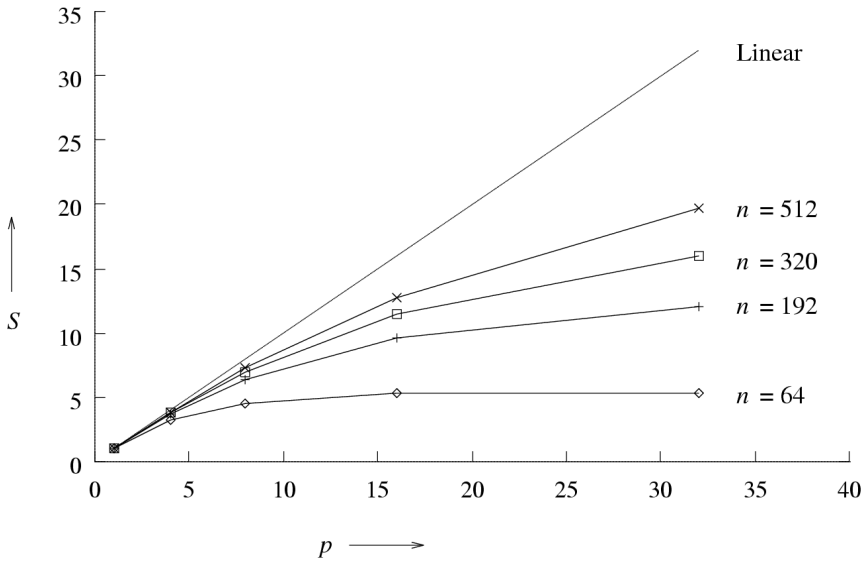


Figure 4.5 Speedup versus the number of processors for adding a list of numbers on a hypercube.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Table 4.1 Efficiency as a function of n and p for adding n numbers on p-processor hypercube.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

n	*p* = 1	*p* = 4	*p* = 8	*p* = 16	*p* = 32
64	1.0	*.80*	.57	.33	.17
192	1.0	.92	*.80*	.60	.38
320	1.0	.95	.87	.71	.50
512	1.0	.97	.91	*.80*	.62

Observation av konstant effektivitet

- E = 0.80 erhålls för
  $p = 4$ och $n = 64$
  $p = 8$ och $n = 192$
  $p = 16$ och $n = 512$

- För samtliga tre fall gäller att $n = 8\ p \log p$

- Vi har tidigare sett att algoritmen är kostnadsoptimal för n = $\Omega(p \log p)$
- Vi fortsätter nu att analysera algorimens prestanda

Skalbarhet

- Förmåga att bibehålla konstant effektivitet då både problemstorlek och antal processorer ökas
- Ett rimligt ytterliggare krav är att mängden nyttjat minne per nod ej ska öka
- Ett skalbart system kan alltid göras kostnadsoptimalt för rätt val av problemstorlek och antal processorer

Vi önskar ett mått på hur mycket problemstorleken måste ökas för att bibehålla effektiviteten då antal processorer ökas

Problemstorlek och overhead

- **W: Problemstorlek** = antalet beräkningssteg då problemet löses på 1 processor
- Vi kommer inledningsvis att anta att ett beräkningssteg tar 1 tidsenhet

  $\Rightarrow W = T_S$

- **T_o: Overhead** kan beskrivas som en funktion av W och p

  $T_O(W,p) = p\,T_P - W$

Overhead i den kostnadsoptimala algoritmen

- Antag att en tidsenhet = $t_a = t_s + t_w$
- n/p tal adderas lokalt på n/p tid
- p delresultat adderas globalt på 2 log p tid
- $T_p = n / p + 2*\log p$
- $T_s = n$
- $T_o = p* T_p - T_s = p(n / p + 2*\log p) - n = 2 p \log p$

Isoeffektivitet

$T_O = p\,T_P - W \quad \Leftrightarrow \quad T_P = (W+T_O)/p$

Detta ger för speedup: $S = \frac{W}{T_P} = \frac{pW}{W+T_O}$

Vi får alltså effektiviteten: $E = \frac{S}{p} = \frac{W}{W+T_O} = \frac{1}{1+T_O/W}$

- Om W hålls konstant och *p* ökas så minskar *E*, ty overhead ökar med p
- Om systemet är skalbart och *W* ökas men *p* hålls konstant så ökar *E*

Isoeffektivitetsfunktionen

- För ett skalbart system kan effektiviteten hållas konstant genom att förhållandet T_O/W hålls konstant
- För önskat värde E gäller: $$E = \frac{1}{1 + T_O / W}$$

$$\frac{T_O}{W} = \frac{1 - E}{E}$$

$$W = \frac{E}{1 - E} T_O$$

- För en konstant $K = E/(1\text{-}E)$ får vi

  $W = K\, T_O(W,p)$

- Detta W kan ofta erhållas som en funktion av p. Denna funktion kallar vi **isoeffektivitetsfunktionen**

Isoeffektivitetsfunktionen för den kostnadsoptimala algoritmen

- $T_o = 2\ p \log p \ \rightarrow W = K * T_o = 2\ K\ p \log p$, dvs den asymptotiska isoeffektivitetsfunktionen är $\Theta(p \log p)$

- För att hålla effektiviteten konstant då #proc ökas från p_1 till p_2, måste problemstorleken ökas från W_1 till W_2 så att $W_2/W_1 = (p_2 \log p_2) / (p_1 \log p_1)$

Noterbart:
Vi såg tidigare att E var konstant = 0.8 för W = 8 p log p. Nu inses sambandet 2 K = 2 E/(1-E) = 2*0.8/0.2 = 8

Overhead beror av fler variabler (än bara p)

- Ofta beror overhead även av W. Då kan det ibland vara omöjligt att bryta ut p ur uttrycket.

Lösning:

- Konstant effektivitet erhålls då T_o/W är konstant
- Om T_o består av flera termer balanserar vi var och en av dem mot W och beräknar isoeffektiviteten med avseende på respektive term
- Den term som kräver störst ökning av W ger den asymptotiska isoeffektiviteten

Kostnadsoptimalt och isoeffektivitet

- Finns något samband?
- Att ett system är kostnadsoptimalt är ekvivalent med att $p\ T_p = \Theta(W)$
- Eftersom $T_p = (W + T_o(W,p))/p$ så är detta

$$W + T_o(W,p) = \Theta(W)$$

$$T_o(W,p) = O(W)$$

$$W = \Omega(T_o(W,p))$$

- Dvs, ett system är **kostnadsoptimalt** om och endast om dess overhead-funktion inte asymptotiskt överskrider dess problemstorlek

Grad av parallellitet och isoeffektivitet

- Grad av parallellitet, $C(W)$ = maximalt antalet operationer som kan utföras parallellt för problemstorlek W
- Isoeffektiviteten m a p parallellitet är optimal $(= \Theta(p))$ endast om graden av parallellitet är $\Theta(W)$
- Om graden av parallellitet är mindre än $\Theta(W)$ så är isoeffektiviteten m a p parallellitet större än $\Theta(p)$

- Den totala isoeffektiviteten är maximum av isoeffektiviteten m a p grad av parallellitet, kommunikation och annan OH

- Givet isoeffektiviteten och tester på ett fåtal processorer kan vi förutsäga systemets uppförande på fler processorer

Mer om overhead

- Givet en overhead-funktion kan vi uttrycka exekveringstid, speedup, effektivitet och kostnad i termer av p och W

Kommunikation

- Om p processorer vardera kommunicerar t_{comm} tidsenheter, ger detta ett bidrag på $p\ t_{comm}$ till OH-funktionen

Dålig lastbalans

- Att någon processor ibland är utan arbete
- Specialfall: Sekventiell del W_s bidrar med pW_s till OH-funkt.

Extra beräkningar

- Om den bästa sekventiella algoritmen kräver W tidsenheter men en bättre parallelliserbar algoritm kräver W', så bör W'-W räknas in i OH-funktionen

Att bestämma minimal exekveringstid

- Givet en funktion $T_p(W,p)$ kan vi finna minimal exekveringstid genom att derivera T_p med avseende på p och sätta derivatan till 0

  $\rightarrow$ antal processorer p_0 som ger minimal exekveringstid

- Genom att sätta $p = p_0$ i $T_p(W,p)$ erhålls den minimala exekveringstiden

Scaled Speedup

[Gustafson, Montry, Benner]

- Speedup som erhålls då problemstorleken *W* ökas linjärt med antal processorer
  Dvs, jämför prestanda för att lösa problem av storlek *W* på 1 proc med prestanda för att lösa problem av storlek *W*p* på *p* processorer
- Relation till iso-effektivitet:
  - om iso-eff är nära linjär så ger scaled speedup relativt lika resultat (också nära linjär)
  - om iso-eff är betydligt sämre så ger scaled speedup och iso-eff mycket olika resultat (de mäter alltså inte samma saker)

Scaled speedup: exempel

T_1: tid för att lösa problem av storlek *W* på 1 processor

T_p: tid för att lösa problem av storlek *p*W* på *p* processorer

$$\text{Scaled speedup} = \frac{p*W/T_p}{W/T_1} = \frac{p*T_1}{T_p}$$

Om det är möjligt att ta fram ett uttryck för *Mflops(.)* har vi:

$$\text{Scaled speedup} = \frac{Mflops(p)}{Mflops(1)}$$

där *Mflops(.)* är prestanda för respektive problemstorlek

Generaliseringar av scaled speedup

[Gustafson et al.], [Worley], [Sun, Ni]

Istället för att skala problemstorleken med en faktor *p*:

- Problemet skalas så att det precis *ryms i minnet* på *p* processorer
- Problem skalas så att exekveringstiden håller sig under ett givet maximum.
  Ger i huvudsak två problemklasser:
  - För vissa problem kan endast ett visst antal processorer nyttjas effektivt
  - För andra problem kan i stort sett godtyckligt stora problem lösas på fixed tid med tillräckligt många processorer

Sizeup

[Sun, Gustafson]

- Vanlig speedup “favoriserar” långsamma processorer och dåligt kodade program
- Sizeup mäter kvoten mellan den problemstorlek som löses på *p* processorer och den som löses på en processor på en given tid

Processor efficiency function

[Chandran, Davis]

- den övre gräns av antal processorer som löser ett problem av storlek W så att exekveringstiden är $\Theta(W/p)$

Data efficiency function

- en typ av "invers" av "processor efficiency function"
- minsta problemstorlek som på ett givet antal processorer ger att exekveringstiden är $\Theta(W/p)$