Scalability, Cost, and Work

Basics of Parallel Computing

Assoc.Prof. Dr. Sascha Hunold

TU Wien

Gustafson-Barsis's Law

Gustafson-Barsis's Law

- Gustafson, 1988, re-evaluation of Amdahl's law
- observation of Gustafson's Law: more powerful computer systems usually solve larger problems, not the same size problem in less time

Gustafson's argument is

- a program consists of serial fraction s and a parallel fraction $1 - s$
- the parallel portion scales perfectly with the number of processors p
- if the parallel part is run on a serial processor, it would take p times longer than on p processors
- the speed-up is then the following (recall that $s + (1 - s) = 1$)

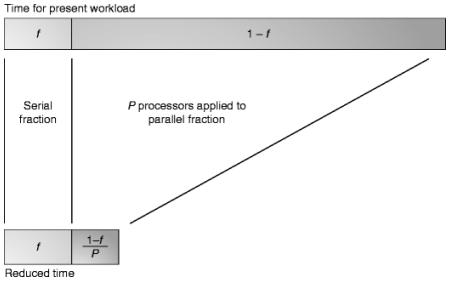
$$S = \frac{s + p(1 - s)}{s + (1 - s)} = s + p(1 - s)$$

- shows that the serial fraction does not theoretically limit the speed-up, if the problem (workload) scales with the number of processors

Amdahl vs. Gutafson

Amdahl

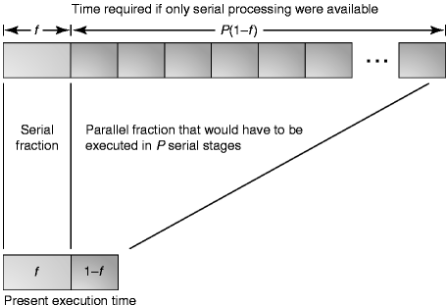
- Amdahl's Law fixes the problem size
- How much faster will my program run with a fixed problem size if I use p processors?



source: Encyclopedia of Parallel Computing

Gustafson

- Gustafson's Law fixes the run time
- How much longer does it take for a given workload to be executed if run sequentially?

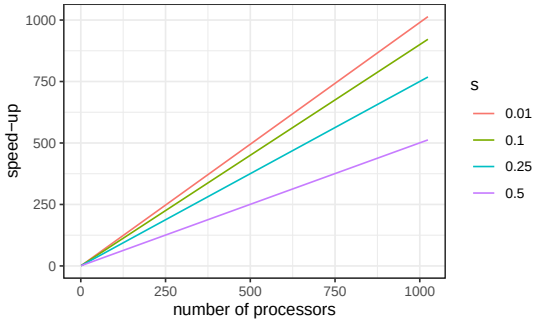


Gutafson's Law Visualized

```r
library(ggplot2)
su <- function(p, seq_share) {
    seq_share + p * (1-seq_share)
}

df <- expand.grid(p = c(1,2,8,16,128,256,1024), seq = c(0.5,0.25,0.1,0.01))
df$su <- su(df$p, df$seq)

ggplot(df, aes(x=p, y=su, color=factor(seq))) + geom_line() +
   xlab("number of processors") + ylab("speed-up") + theme_bw() + labs(color="s")
```



Strong and Weak Scaling

Strong and Weak Scaling

- in parallel computing, we typically use two analyses (depending on the type of problem)

Strong Scaling (based on Amdahl's law)

- keep input size fixed
- increase the number of processors

Weak Scaling (based on Gustafson's law)

- increase the number of processors
- scale instance size with the number of processors

Strong and Weak Scaling Example I

- we perform a weak scaling and strong scaling experiment with the same code
- we use our example from the previous lecture, the `dummy_count` in R

```
1 dummy_count <- function(max) {
2   sum = 0
3   for(i in 1:max) {
4     sum = sum + i
5   }
6   sum
7 }
```

Strong and Weak Scaling Example II

- in the strong-scaling experiment, we keep the problem size fixed
- thus, we fix `input <- 1:10000`
- thus, the first ten elements of `input` are
  - `1 2 3 4 5 6 7 8 9 10`
- then, we increase the number of processors (cores)

```
for (i in p) {
  a <- within(benchmark(replications=c(10),
                        mclapply(input, dummy_count, mc.cores = i),
                        columns=c('test', 'elapsed', 'replications')),
              { average = elapsed/replications })
  df[df$p==i, ]$tstrong <- a$average[[1]]
}
```

- we can then compute the relative speed-up
  - as usual $T_{\text{par}}(n,1)/T_{\text{par}}(n,p)$
  - `df$sustrong <- df[df$p==1,]$tstrong / df$tstrong`

Strong and Weak Scaling Example III

- in the weak-scaling experiment, we keep the problem size per core fixed
  - thus, we grow the problem with an increasing number of cores
- with 1 core, we use `input <- 1:10000`
  - with 2 cores, we double the work to `c(input, input)`
  - with 3 cores, we triple the work to `c(input, input, input)`
- the experiment then looks like this

```
for (i in p) {
  a <- within(benchmark(replications=c(10),
                        mclapply(rep(input,i), dummy_count, mc.cores = i),
                        columns=c('test', 'elapsed', 'replications')),
              { average = elapsed/replications })
  df[df$p==i, ]$tweak <- a$average[[1]]
}
```

- we compute a scaled speed-up, where we relate the time to finish the scaled workload on only 1 processor to the time it took to complete the scaled workload on p processors
- $S = \frac{pT_{\text{par}}(n,1)}{T_{\text{par}}(pn,p)}$
- `df$suscaled <- (df[df$p==1,]$tweak * df$p) / df$tweak`

Strong and Weak Scaling Example IV

hydra35

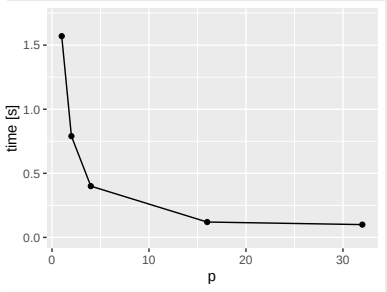
- we now perform this experiment on one compute node of the `hydra` cluster
- we select `hydra35` (no particular reason, but we had to select one node)
- the experimental results are as follows

p	tstrong	tweak	sustrong	suscaled
1	1.57	1.57	1	1
2	0.79	1.59	1.98	1.98
4	0.4	1.59	3.91	3.94
16	0.12	1.63	13.18	15.38
32	0.1	1.7	16.24	29.54

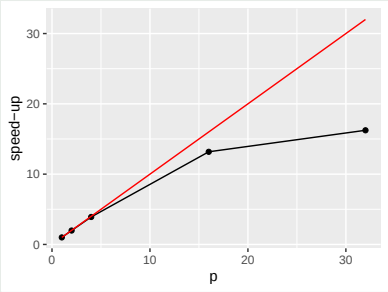
Strong and Weak Scaling Example V

Strong Scaling - hydra35

```
ggplot(df, aes(x=p, y=tstrong)) +
  geom_point() + geom_line() +
  ylim(0,1.7) +
  ylab("time [s]")
```



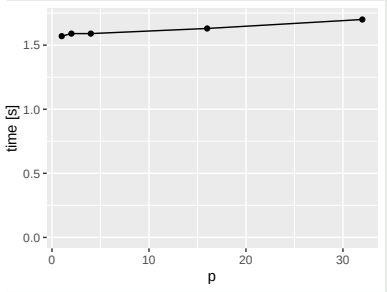

```
ggplot(df, aes(x=p, y=sustrong)) +
  geom_point() + geom_line() +
  geom_line(data=df, aes(x=p, y=p), color="red") +
  ylab("speed-up")
```



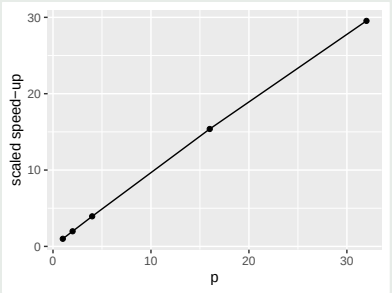
Strong and Weak Scaling Example VI

Weak Scaling - hydra35

```
ggplot(df, aes(x=p, y=tweak)) +
  geom_point() + geom_line() +
  ylim(0,1.7) +
  ylab("time [s]")
```




```
ggplot(df, aes(x=p, y=suscaled)) +
  geom_point() + geom_line() +
  ylab("scaled speed-up")
```



A Note on Weak Scaling I

- we usually check whether the runtime stays constant when increasing both the problem size and the number of processors
- but we need to scale the problem size correctly
- in our previous example
  - the number of operations depends on the values in the list
  - for a list `1:n` the total number of operations is $O = \sum_{j=1}^{n} j$
    - there are j additions per invocation of `dummy_count`
  - with p processors, we simply multiply with p, i.e., pO

A Note on Weak Scaling II

- but what if we do a matrix-vector product?
  - complexity of a matrix vector product $O(n^2)$, for a matrix of size $n \times n$
- assume that we parallelize the matrix-vector product
  - each processor gets n/p rows of matrix A
  - each processor needs to perform $2n^2/p$ operations (ADD + MUL per element), which is its work
- if we want to increase n with p, we solve by n
  - $w_1 = w_p = \frac{2n^2}{p}$, $n = \sqrt{\frac{w_1 p}{2}}$
  - w_1 denotes the work done on one processor for the sequential run
  - w_p denotes the work done per processor in the parallel run
- now, we want to perform a weak-scaling analysis of our algorithm
  - we set $n = 100$, then $w_1 = 2n^2 = 20\,000$

```
1 n <- 100
2 w <- 2*n^2
3 df <- data.frame(p=c(1,2,4,8,16,32,64,128))
4 df$n <- ceiling(sqrt(w*df$p / 2))
5 print(toOrg(df))
```

p	n
1	100
2	142
4	200
8	283
16	400
32	566
64	800
128	1132

Embarrassing Parallelism

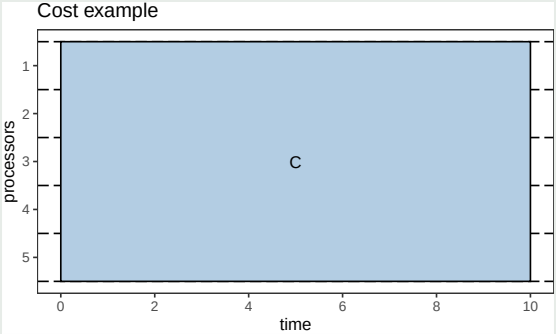
- We often hear the word "embarrassingly parallel", but what does that mean?
- it means that the parallelization strategy is obvious
  - no advanced algorithms needed
  - very little communication overhead
  - little or no effort to split up work
- for example: parameter sweeps
  - program A is called with parameters p_1 and p_2
  - p_1 may have 30 different values
  - p_2 may have 20 different values
  - we got 600 different combinations of p_1 and p_2
  - we can parallelize easily over these combinations
  - Note: we could also parallelize A itself

Cost and Work of Parallel Programs

Cost and Work of Parallel Programs I

Cost

The cost of a parallel algorithm denotes the total time that one uses all p processors, i.e., $C = pT_{\text{par}}(p, n)$.



Cost and Work of Parallel Programs II

Cost Optimality [1]

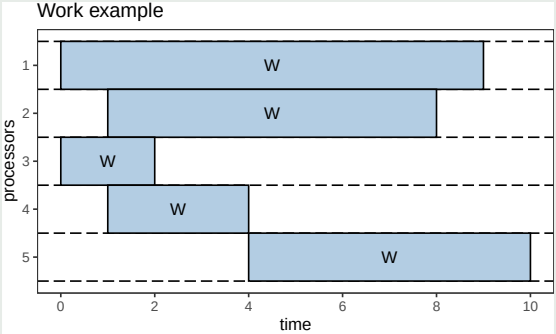
A parallel program is cost optimal if the cost of solving a problem on a parallel computer has the same asymptotic growth (as a function of the input size) as the fastest-known sequential algorithm on a single processing element, i.e.,

$$pT_{\text{par}}(p,n) \in O(T_{\text{seq}}(n)) \quad .$$

Cost and Work of Parallel Programs III

Work

The work W of a parallel algorithm denotes the number of operations that are carried out.



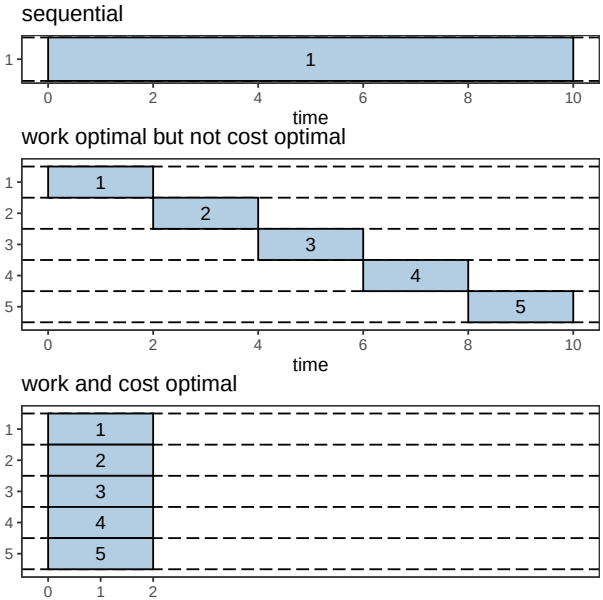
Cost and Work of Parallel Programs IV

Work Optimality

A parallel program is work optimal if the work W has the same asymptotic growth as the fastest-known sequential algorithm on a single processing element, i.e.,

$$W \in O(T_{\text{seq}}(n)) \quad .$$

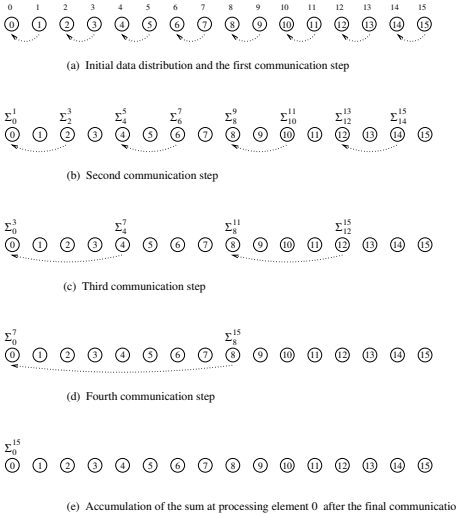
Cost and Work of Parallel Programs V



Cost Analysis / Example

Example of Cost Analysis - Attempt 1 I

- we consider the problem of adding n numbers in parallel
- we assume to have n processing elements (yes, as many processing elements as numbers) and we also assume that n is a power of two
- we can solve the problem in $\log n$ steps
  - propagate partial sums up a logical binary tree



source: [1]

Example of Cost Analysis - Attempt 1 II

- sequential running time $\Theta(n)$
- parallel running time $\Theta(\log n)$
- speed-up: $\Theta\left(\frac{n}{\log n}\right)$

```
1 n <- c(2,4,16,128,1024, 1024^2)
2 df <- data.frame(n=n, su= n / log2(n))
3 df
```

```
        n          su
1       2     2.00000
2       4     2.00000
3      16     4.00000
4     128    18.28571
5    1024   102.40000
6 1048576 52428.80000
```

Example of Cost Analysis - Attempt 1 III

What about the cost?

- we use n processors for time $\Theta(\log n)$
- thus, the cost of the parallel algorithm is $\Theta(n \log n)$
- recall that the sequential running time is $\Theta(n)$
- we ask if $O(n \log n) \in O(n)$?
  - no, and thus, the algorithm is not cost-optimal

Attempt 2 - Scaling I

Can we improve this algorithm?

- using n processors for n elements unrealistic
  - although good to design algorithm
  - we can always scale down a parallel system: some processors simulate the work of other processors
- let us assume now that $p < n$
  - n and p are powers of two
- we use the same algorithm as before, but we simulate the previous n processors on p processors
- our previous processor i is now mapped to processor $i \mod p$
  - e.g. $n = 16$, $p = 4$

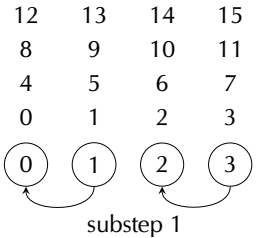
```
1 df <- data.frame(i=seq(0,15))
2 df$p_real <- df$i %% 4
3 df
```

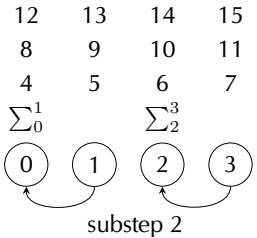
```
    i p_real
1   0      0
2   1      1
3   2      2
4   3      3
5   4      0
6   5      1
7   6      2
8   7      3
9   8      0
10  9      1
11 10      2
12 11      3
13 12      0
14 13      1
15 14      2
16 15      3
```

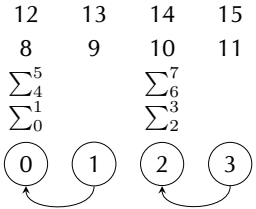
Attempt 2 - Scaling II

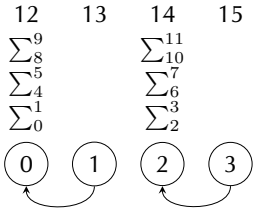
1. virtual processor i is simulated by physical processor $i \mod p$
2. in the first $\log p$ steps of the original algorithm on n processors now take $(n/p) \log p$ steps on p processors
3. in the remaining steps, no communication required (data already at physical processor), data added locally
4. our example, with $n = 16$ and $p = 4$

Attempt 2 - Scaling (Visualization)



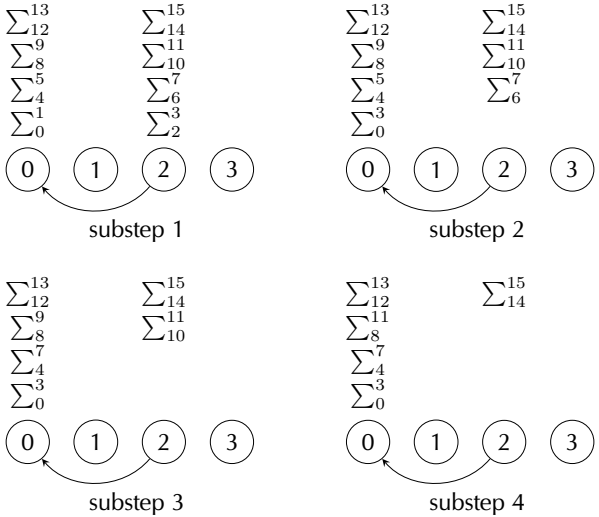






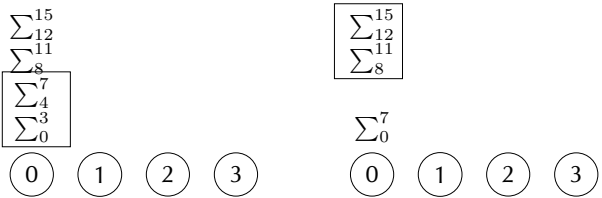
- simulating 16 original processors on 4 processors
- first communication step
- original algorithm
  - p1 $\rightarrow$ p0
  - p3 $\rightarrow$ p2
  - p5 $\rightarrow$ p4
  - p7 $\rightarrow$ p6
  - ...
- now, p0 simulates i $\mod 4$
  - it simulates p0, p4, p8, p12
  - thus, it needs 4 substeps
- similarly for the other processors

Attempt 2 - Scaling (Visualization)



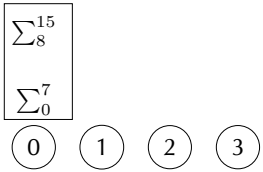
- simulating 16 original processors on 4 processors
- second communication step
- original algorithm
  - p2 $\rightarrow$ p0
  - p4 $\rightarrow$ p2
  - p6 $\rightarrow$ p4
  - p8 $\rightarrow$ p6
  - ...
- now, p0 simulates p0, p4, p8, p12
  - p2, simulates p2, p6, p10, p14
  - thus, we need 4 substeps

Attempt 2 - Scaling (Visualization)



- subresults are now locally available on p0
- no further communication required

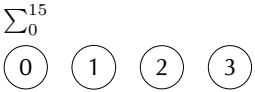
Attempt 2 - Scaling (Visualization)



substep 1

- subresults are now locally available on p0
- no further communication required

Attempt 2 - Scaling (Visualization)



substep 1

- final result
- runtime of original algorithm: $\log n = \log 16 = 4$
  - cost: $n \log n$
- runtime of simulation:
  - communication: $n/p \log p = 16/4 \log 4 = 4 \cdot 2 = 8$
  - computation: $n/p = 16/4 = 4$
  - overall: $O(n/p \log p + n/p) = O(n/p \log p)$
  - cost: $O(p \frac{n}{p} \log p) = O(n \log p)$
- thus, this strategy is also not cost-optimal

Attempt 3 - First Things First

cost-optimal algorithm

- how about adding the n/p numbers first locally by the p processors
  - this takes time $O(n/p)$
- now, we have p numbers to add, each number on 1 of the p processors
- we have shown before that this can be done on $O(\log p)$
- therefore, this resulting running time is $O(n/p + \log p)$
- the cost (running time multiplied by p) is
  - $O(n + p \log p)$
- if $n \in \Omega(p \log p)$, the cost is $O(n)$, which is the same as in the sequential case
  - thus, the algorithm is cost-optimal

References

[1] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Second. Addison-Wesley, 2003. ISBN: 0201648652 9780201648652.