Analytical Modeling of Parallel Systems and Performance Analysis

Topic Overview

- Sources of Overhead in Parallel Programs
- Performance Metrics for Parallel Systems
- Effect of Granularity on Performance
- Scalability of Parallel Systems

Analytical Modeling - Basics

- A sequential algorithm is evaluated by its runtime
  - In general, asymptotic runtime as a function of input size
- The asymptotic runtime is independent of the platform.
  - Analysis “at a constant factor”.
- A parallel algorithm has more parameters.

Big O notation – O(g(n))

- Big O notation is used to describe the performance or complexity of an algorithm.

- Big O specifically describes the bounds
  - can be thought of as **worst-case** scenario
  - can be used to describe the execution time required by an algorithm.

O(1)

- O(1) describes an algorithm that will always execute in the same time regardless of the size of the input data set.

```
bool IsFirstElementNull(String[] strings)
 {
  if(strings[0] == null)
      return true;
  return false;
}
```

O(n)

- O(n) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
bool ContainsValue(String[] strings, String value)
{
  for(int i = 0; i < strings.Length; i++)
  {
       if(strings[i] == value)
              return true;
  }
  return false;
}
```

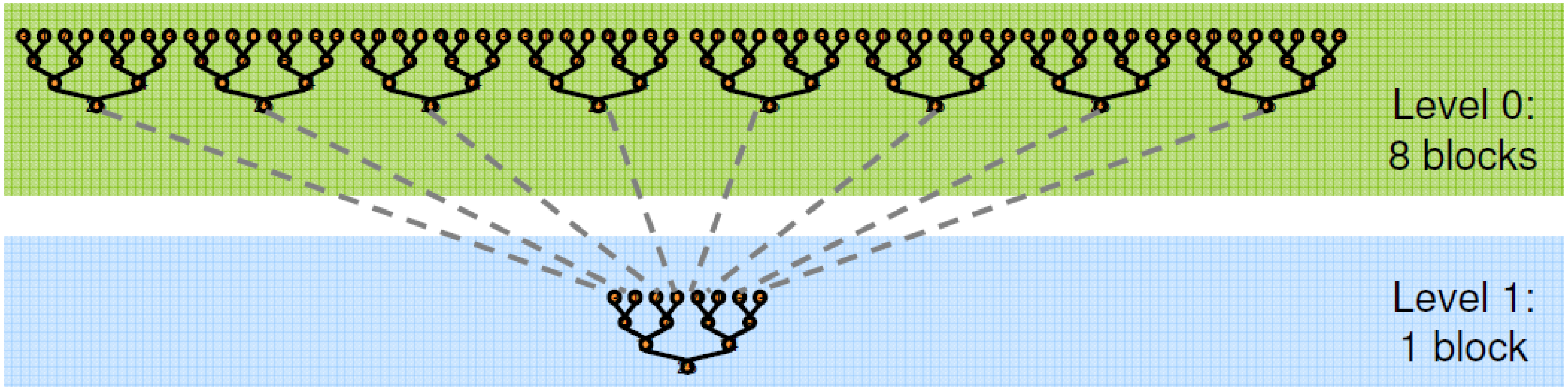
O(n^2)

- O(n^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(n^3), O(n^4) etc.

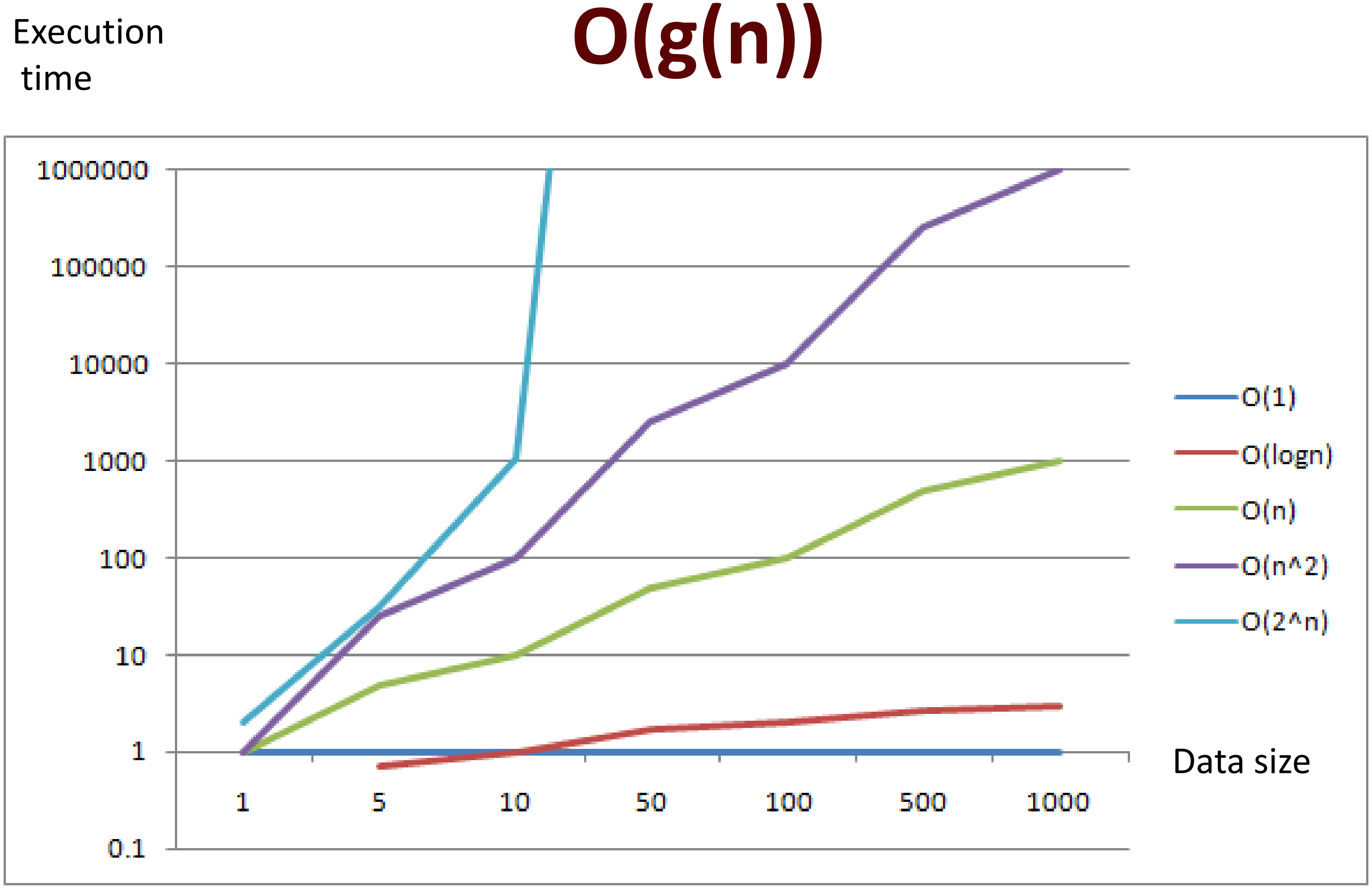
```
bool ContainsDuplicates(String[] strings)
{
   for(int i = 0; i < strings.Length; i++)
   {
         for(int j = 0; j < strings.Length; j++)
         {
                 if(i == j) // Don't compare with self
                          continue;
                 if(strings[i] == strings[j])
                          return true;
         }
   }
   return false;
}
```

O(log n)

- Algorithms iteratively halving the datasets as we have seen before.
- Example: Parallel Reduction



O(g(n))



Execution time
1000000
100000
10000
1000
100
10
1
0.1
1
5
10
50
100
500
1000
Data size
O(1)
O(logn)
O(n)
O(n^2)
O(2^n)

Big-Theta (Θ) and Big-Omega (Ω)

- $f(n) \in O(g(\boldsymbol{n}))$:
  - *f is bounded above by g asymptotically*
  - (worst case scenario)

- $f(n) \in \Omega(g(\boldsymbol{n}))$:
  - *f* is bounded below by *g* asymptotically
  - (best case scenario)
- $f(n) \in \Theta(g(\boldsymbol{n}))$:
  - *f* is bounded both above and below

Analytical Modeling - Basics

- The parallel runtime of a program depends on the:
  - input size,
  - the number of processors,
  - the communication parameters of the machine.
- An algorithm must therefore be analyzed in the context of the underlying platform.
- A parallel system is a combination of:
  - a parallel algorithm
  - an underlying platform.

Analytical Modeling - Basics

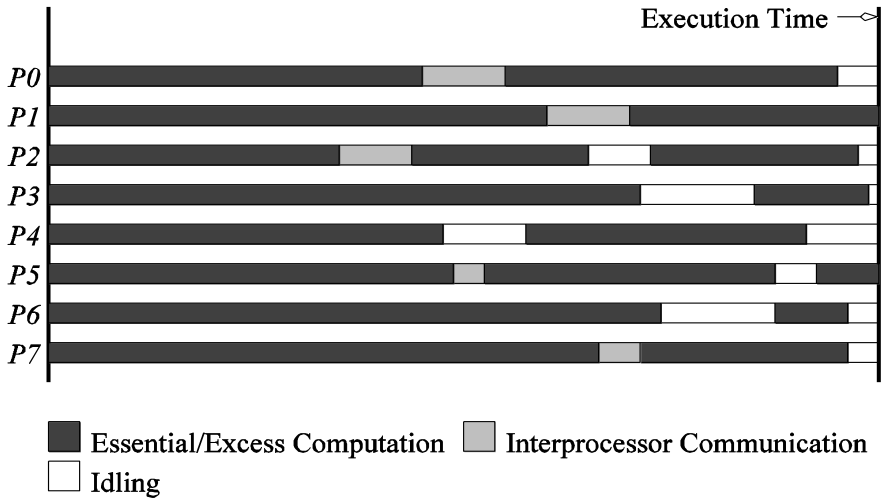
- A number of performance measures are intuitive.
- Execution time: the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble (T_P).
- But how does this scale when the number of processors is changed and the program is ported to another machine altogether?
- How much faster is the parallel version?
- This brings the obvious follow up question:

  "What's the baseline serial version with which we compare?"

Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run twice as fast?

  No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.



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.

Sources of Overheads in Parallel Programs

- **Inter-process interactions**: Processors working on any non-trivial parallel problem will need to talk to each other.

- **Idling**: Processes may idle because of load imbalance, synchronization, or serial components.

- **Excess Computation**: This is computation not performed by the serial version.
  - This might be because the serial algorithm is difficult to parallelize,
  - or that some computations are repeated across processors to minimize communication.

Performance Metrics for Parallel Systems: Execution Time

- Serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.

- The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.

- We denote the serial runtime by T_S and the parallel runtime by T_P .

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.

$$T_{all} = p\, T_P \qquad (p \text{ is the number of processors}).$$

- Observe that $T_{all} - T_S$ is the total time spend by all processors combined in non-useful work. This is called the *total overhead* (T_o).

$$T_o = p\, T_P - T_S$$

Performance Metrics for Parallel Systems: Speedup

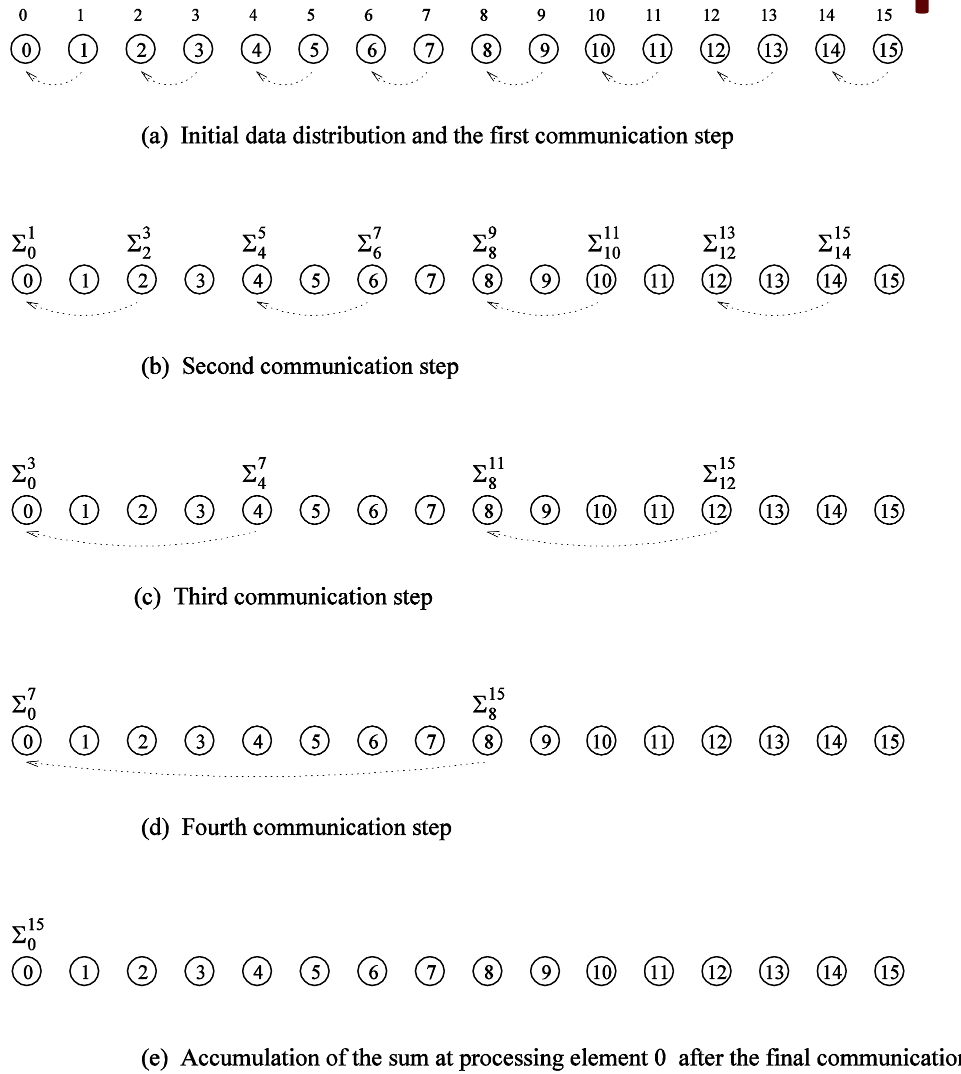
- What is the benefit from parallelism?
- Speedup (***S***) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

Performance Metrics: Example

- Consider the problem of adding ***n*** numbers by using ***n/2*** processing elements.

- If ***n*** is a power of two, we can perform this operation in $\log_2 n$ steps by parallel reduction

Performance Metrics: Example



(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 processing element 0 after the final communicatio

Computing the global sum of 16 partial sums using 8 processing elements .

Performance Metrics: Example

- We have the parallel time
  $T_P = \Theta(\log_2 n)$

- We know that $T_S = \Theta(n)$

- Speedup S is given by $S = \Theta(n / \log_2 n)$

Performance Metrics: Speedup

- For a given problem, there might be many serial algorithms available.

- These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.

- For the purpose of computing speedup, we always consider **the best sequential program** as the baseline.

Performance Metrics: Speedup Example

Consider the problem of parallel bubble sort.

- The serial time for bubble sort is 150 seconds.

- The parallel time for odd-even sort (efficient parallelization of bubble sort) is 40 seconds.
  - The speedup would appear to be 150/40 = 3.75.

- What if another serial quicksort implementation only took 30 seconds?
  - In this case, the speedup is 30/40 = 0.75.

Performance Metrics: Speedup Bounds

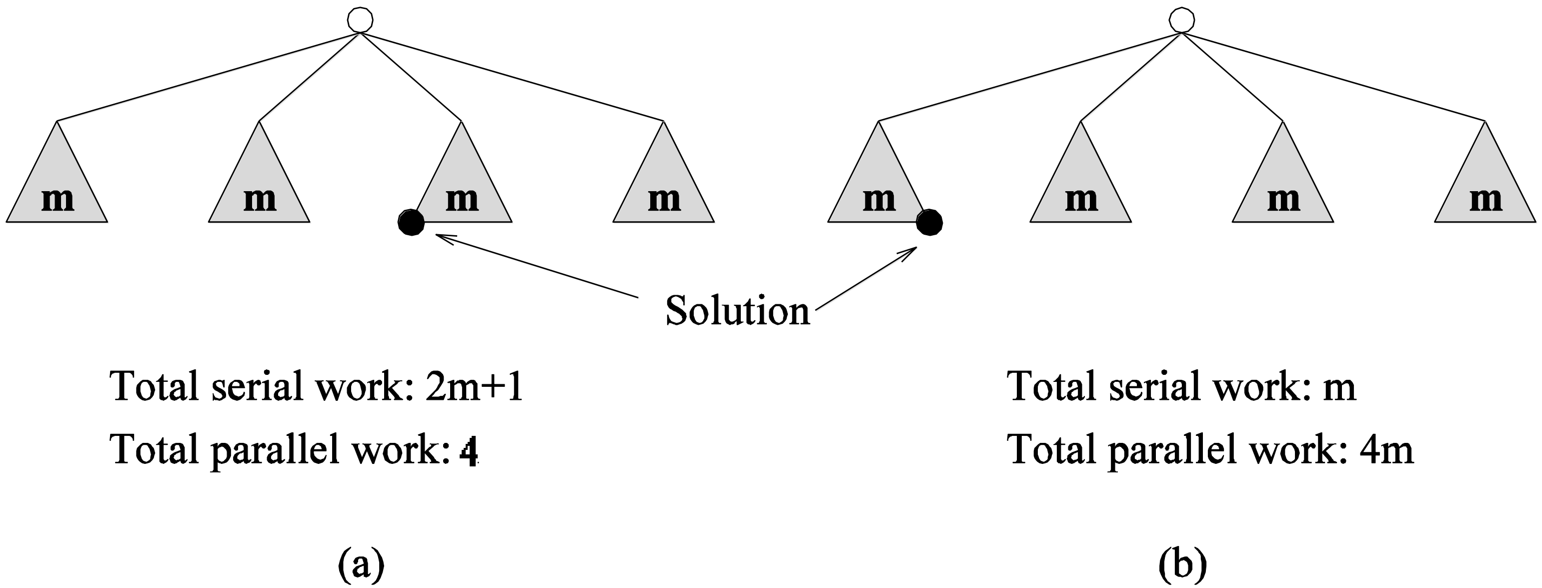
- Speedup can be as low as 0 (the parallel program never terminates).
- Speedup, in theory, should be upper bounded by p
  - we can only expect a p-fold speedup if we use p processing elements.

Performance Metrics: Speedup Bounds

- A speedup greater than p is possible only if each processing element spends less than time T_S / p solving the problem.
  - In this case, a single processor could be time-slided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Superlinear Speedups

One reason for superlinearity is that the parallel version does less work than corresponding serial algorithm.



(a) Shows super-linear behavior while (b) shows sub-linear behavior

Performance Metrics: Superlinear Speedups

Resource-based superlinearity:

The higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore superlinearity.

Example:

- A processor with 64KB of cache yields an 80% hit ratio, the remaning comes from local memory.
- If two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory.

  If DRAM access time is 100 ns, cache access time is 2 ns, and remote memory access time is 400ns,

Case1: 2*0.8+100*0.2=21.6 ns

Case 2: 2*0,9+100*0,08+400*0,02=17,8 ns

This corresponds to a speedup of 1.21 in memory access.

Performance Metrics: Superlinear Speedups

Example:
DRAM access time is 100 ns
cache access time is 2 ns
remote memory access time is 400ns

Case1: A processor with 64KB of cache yields an 80% hit ratio, the remaning comes from local memory.

$$2*0.8+100*0.2=21.6 \text{ ns}$$

Case 2: Two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory.

$$2*0,9+100*0,08+400*0,02=17,8 \text{ ns}$$

This corresponds to a speedup of 1.21 in memory access.

Performance Metrics: Efficiency

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed

- Mathematically, it is given by

$$E = \frac{S}{p}. \quad , 0 \leq E \leq 1$$

Performance Metrics: Efficiency Example

- The speedup of adding numbers on *n* processors is given by

$$S = \frac{n}{\log n}$$

- Efficiency is given by

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

Parallel Time, Speedup, and Efficiency Example

Consider the problem of filtering images.

The problem requires us to apply a template to each pixel.

random image I(x,y)

8	8	2	2	12
1	3	4	7	7
3	15	5	9	5
3	1	9	12	12
1	3	15	4	15

averaging filter W(x,y)

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

8/9 + 8/9 + 2/9 + 1/9 + 3/9 + 4/9 + 3/9 + 15/9 + 5/9 = 5.3

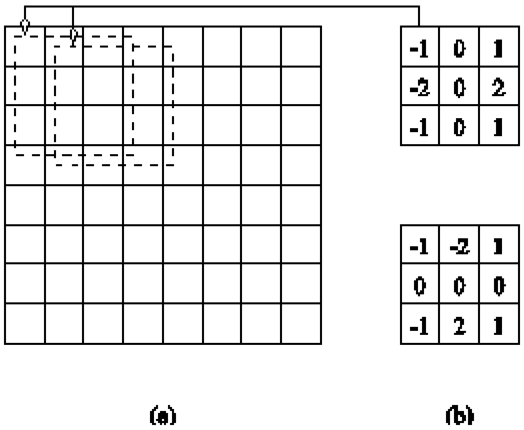
filtered image I'(x,y)

0	0	0	0	0
0	5			0
0				0
0				0
0	0	0	0	0

Parallel Time, Speedup, and Efficiency Example

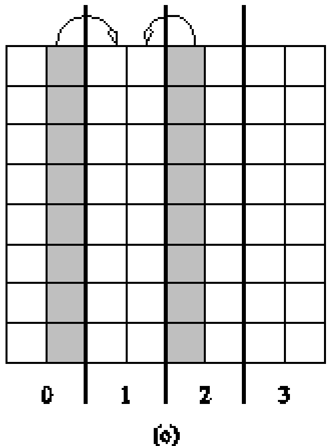
Edge-detection problem requires us to apply a ***3*** x ***3*** template to each pixel.

If each multiply-add operation takes time t_c, the serial time for an $n \times n$ image is given by $T_S = 9\, t_c\, n^2$.



Parallel Time, Speedup, and Efficiency Example (continued)

- One possible parallelization partitions the image equally into vertical segments, each with n^2 / p pixels.



- The boundary of each segment is $2n$ pixels. This is also the number of pixel values that will have to be communicated. This takes time $2(t_s + t_w n)$.
- Templates may now be applied to all n^2 / p pixels in time

$T_S = 9\, t_c n^2 / p$.

Parallel Time, Speedup, and Efficiency Example (continued)

- The total time for the algorithm is therefore given by:

$$T_P = 9t_c\frac{n^2}{p} + 2(t_s + t_w n)$$

- The corresponding values of speedup and efficiency are given by:

$$S = \frac{9t_c n^2}{9t_c\frac{n^2}{p} + 2(t_s + t_w n)}$$

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

Cost of a Parallel System

- Cost is the product of parallel runtime and the number of processing elements used ($p\,T_P$).
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.
- Since $E = T_S / p\,T_P$, for cost optimal systems, $E = O(1)$.

Cost of a Parallel System: Example

Consider the problem of adding *n* numbers on *p* processors. Assuming $\boldsymbol{p = n}$

- We have, $\boldsymbol{T_P = \log n}$
- The cost of this system is given by $\boldsymbol{p\,T_P = n \log n}$.
- Since the serial runtime of this operation is $\Theta(\boldsymbol{n})$,

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

➔ the algorithm is not cost optimal.

Impact of Non-Cost Optimality

Consider a sorting algorithm that uses n processing elements to sort the list in time:

$T_P = (\log n)^2$

Serial runtime of a (comparison-based) sort is

$T_S = n \log n$

Then;

- Speedup: $S = n / \log n$
- Efficiency: $E = 1 / \log n$
- Cost $C = n (\log n)^2$.

This algorithm is not cost optimal but only by a factor of ***log n***.

If $p < n$, assigning n tasks to p processors gives:

- $T_P = n (\log n)^2 / p$.
- $S = p / \log n$.
- This speedup goes down as the problem size n is increased for a given p !

Effect of Granularity on Performance

- Often, using fewer processors improves performance of parallel systems.
- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down* a parallel system.
- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scaled down processors.

Amdahl's law

Limitations of inherent parallelism: a part *s* of the algorithm is not parallelizable

$$\boxed{T_{seq} = \underbrace{(1-s).T_{seq}}_{\text{parallelizable}} + \underbrace{s.T_{seq}}_{\text{not parallelizable}}}$$

$$\boxed{T_{par} = \frac{(1-s).T_{seq}}{p} + s.T_{seq}}$$

$$\boxed{Speedup_{\max} = \frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{\frac{(1-s).T_{seq}}{p} + s.T_{seq}} = \frac{p}{1+(p-1).s}}$$

Assume no other overhead

$$\text{Speedup} < \frac{p}{1+(p-1).s}$$

$$\text{Efficiency} < \frac{1}{1+(p-1).s}$$

If *p* is big enough:

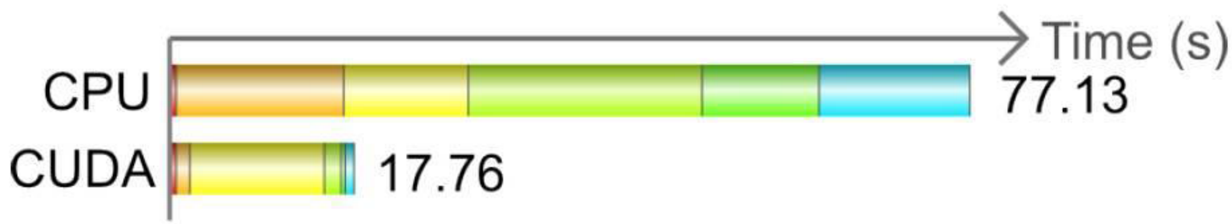
$$\text{Speedup} < \frac{1}{s}$$

s	Speedup_{max}
10%	10
25%	4
50%	2
75%	1.33

Amdahl example: video decoding

Decoding 1080p video sequence

Stage	CPU (s)	CUDA (s)	
1 MOTION_DECODE	0.64	0.64	
2 MOTION_RENDER	16.16	1.33	12 ×
3 RESIDUAL_DECODE	12.00	12.94	
4 WAVELET_TRANSFORM	22.52	1.63	14 ×
5 COMBINE	11.27	0.39	29 ×
6 UPSAMPLE	14.53	0.85	17 ×
Total	77.13	17.76	4.3 ×

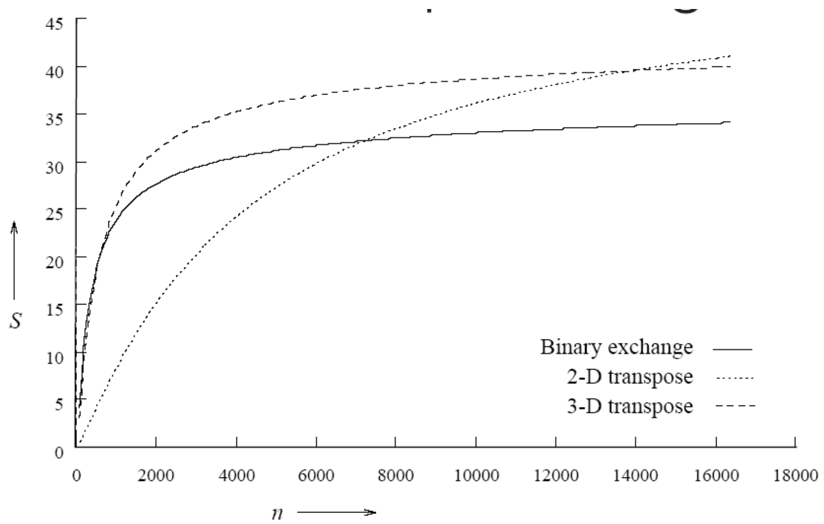


Wladimirvan der Laan, University of Groningen

Scalability of Parallel Systems

How do we extrapolate performance from small problems and small systems to larger problems on larger configurations?

Consider three parallel algorithms for computing an ***n***-point Fast Fourier Transform (FFT) on 64 processing elements.



A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$.

Clearly, it is difficult to infer scaling characteristics from observations on small datasets on small machines.

Scaling Characteristics of Parallel Programs

- The efficiency of a parallel program can be written as:

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

or

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

- Derived from overhead function which is $\boldsymbol{T_0 = pT_p - T_s}$
- The total overhead function $\boldsymbol{T_o}$ is an increasing function of $\boldsymbol{p}$. This is because every program must contain some serial component. If this serial component of the program takes time $\boldsymbol{t_{serial}}$, then during this time all the other processing elements must be idle. This corresponds to a total overhead function of $\boldsymbol{(p-1)t_{serial}}$.

Scaling Characteristics of Parallel Programs

- For a given problem size (i.e., the value of T_S remains constant), as we increase the number of processing elements, T_o increases.

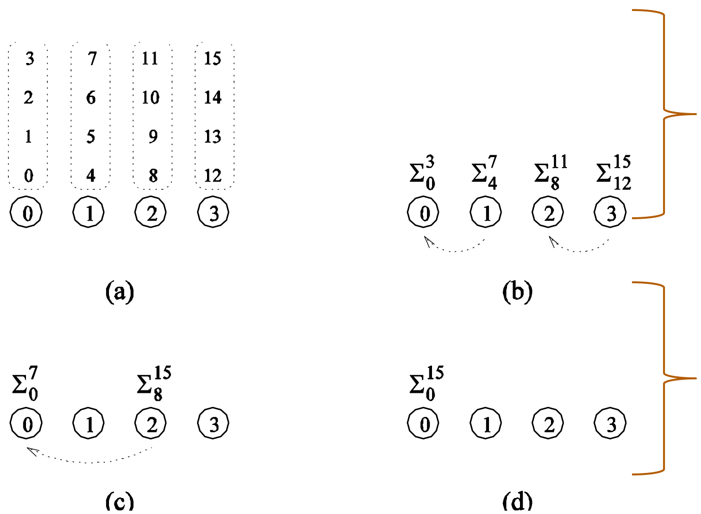
- The overall efficiency of the parallel program goes down. This is the case for all parallel programs.

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

Scaling Characteristics of Parallel Programs:

- Consider the problem of adding ***n*** numbers on ***p*** processing elements. Assume unit time for adding two numbers.
- We have seen that:

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



Θ (***n*** / ***p***).

Θ(**log *p***).

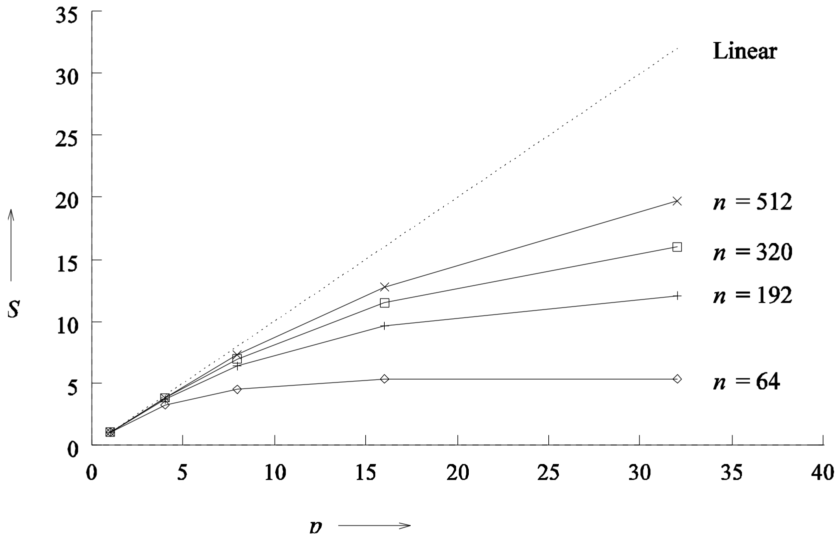
$$S = \frac{n}{\frac{n}{p} + 2\log p}$$

$$E = \frac{1}{1 + \frac{2p\log p}{n}}$$

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***.

Scaling Characteristics of Parallel Programs: Example (continued)

Plotting the speedup for various input sizes gives us:



- Speedup versus the number of processing elements for adding a list of numbers.
- Speedup tends to saturate and efficiency drops as a consequence of Amdahl's law.
- 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***.

Scaling Characteristics of Parallel Programs

- Total overhead function T_o is a function of both problem size T_s and the number of processing elements p. In many cases, T_o grows sublinearly with respect to T_s.
- In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant.
- For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant.
- We call such systems *scalable* parallel systems.

n	p=1	p=4	p=8	p=16	p=32
64	1	0.8	0.57	0.33	0.17
192	1	0.92	0.8	0.6	0.38
320	1	0.95	0.87	0.71	0.5
512	1	0.97	0.91	0.8	0.62

Scaling Characteristics of Parallel Programs

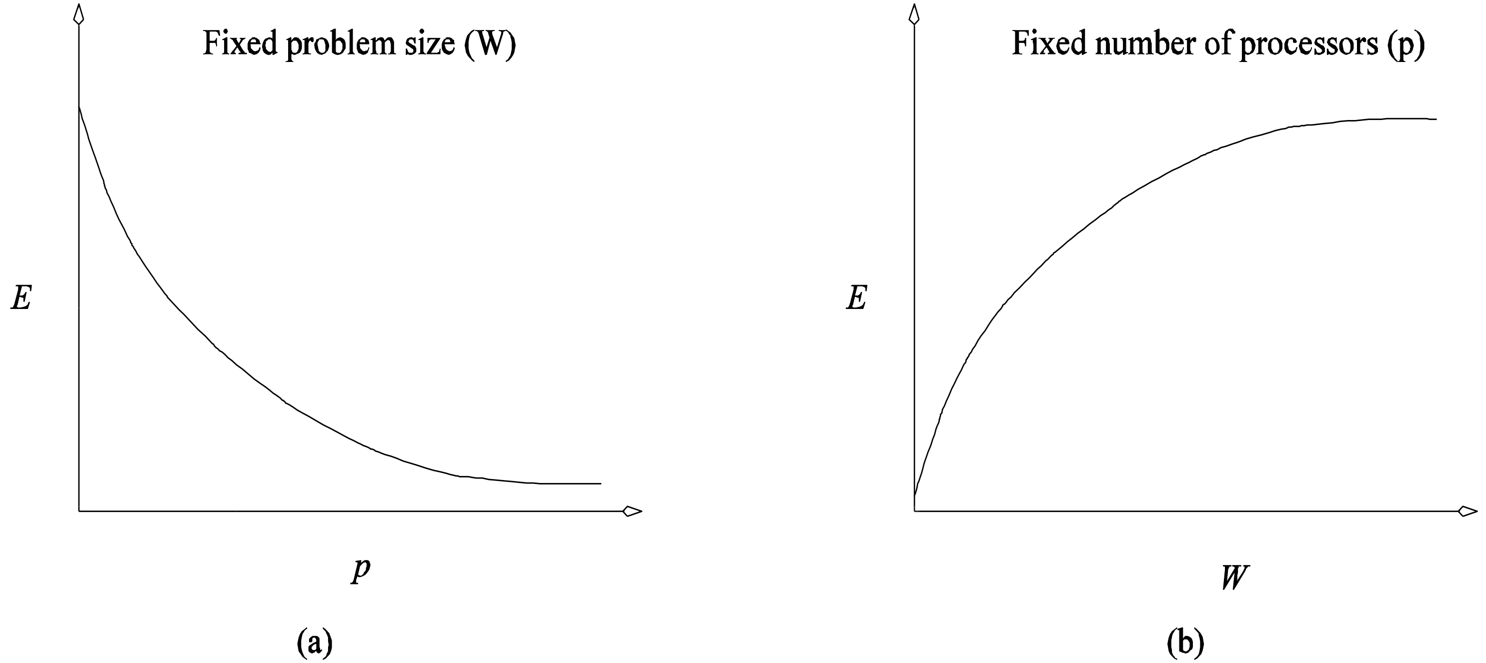
- Recall that cost-optimal parallel systems have an efficiency of $\Theta(1)$.
- Scalability and cost-optimality are therefore related.
- A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately.

Isoefficiency Metric of Scalability

- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.

- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

Isoefficiency Metric of Scalability



Variation of efficiency:

(a) as the number of processing elements is increased for a given problem size;

(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-

Isoefficiency Metric of Scalability

- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?
- This rate determines the scalability of the system.
- Before we formalize this rate, we define the problem size ***W*** as the number of basic computation steps in the best serial algorithm to solve the problem on a single processing element.

Isoefficiency Metric of Scalability

- We can write parallel runtime as:

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

The resulting expression for speedup is

$$S = \frac{W}{T_P}$$

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

- Finally, we write the expression for efficiency as

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

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

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

Isoefficiency Metric of Scalability

- For scalable parallel systems, efficiency can be maintained at a fixed value if the ratio T_o / W is maintained at a constant value.
- For a desired value E of efficiency,

$$\begin{aligned} E &= \frac{1}{1 + T_o(W,p)/W}, \\ \frac{T_o(W,p)}{W} &= \frac{1-E}{E}, \\ W &= \frac{E}{1-E} T_o(W,p). \end{aligned}$$

- If $K = E / (1 - E)$ is a constant depending on the efficiency to be maintained, since T_o is a function of W and p, we have

$$W = KT_o(W,p).$$

Isoefficiency Metric of Scalability

- The problem size ***W*** can usually be obtained as a function of ***p*** by algebraic manipulations to keep efficiency constant.

- This function is called the *isoefficiency function*.

- This function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements.

Isoefficiency Metric: Example

- The overhead function for the problem of adding ***n*** numbers on ***p*** processing elements is approximately ***2p* log *p*** .

- Substituting T_o by ***2p* log *p*** , we get

$$W = K2p \log p.$$

Thus, the asymptotic isoefficiency function for this parallel system is

$$\Theta(p \log p)$$

- If the number of processing elements is increased from ***p*** to ***p'***, the problem size must be increased by a factor of

(***p'* log *p'***) / (***p* log *p***)

to get the same efficiency as on ***p*** processing elements.

Reading List

- “Introduction to Parallel Computing”, 2nd Edition, 2003, Addison Wesley
  - By Ananth Grama; Anshul Gupta; George Karypis; Vipin Kumar
- Chapter 5: Analytical Modeling of Parallel Systems
  - http://proquestcombo.safaribooksonline.com/0201648652/ch05

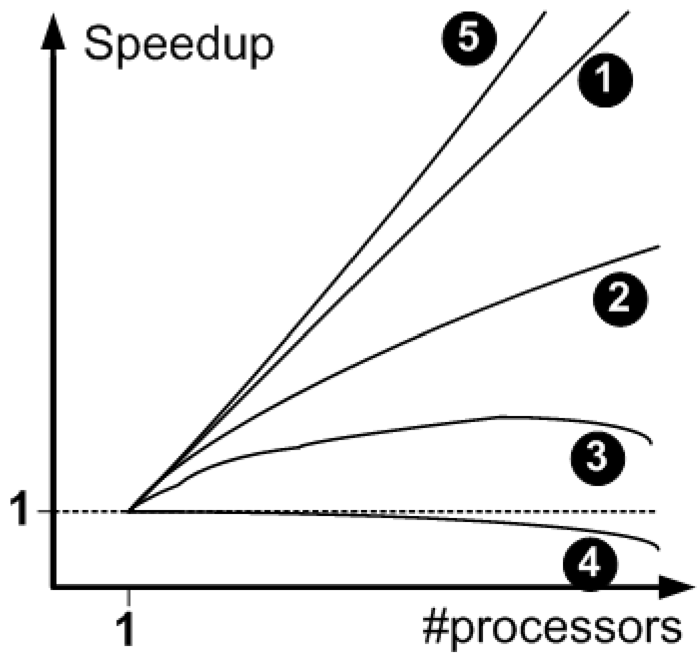
Performance Analysis

Slides adapted from:
Parallel Systems: Performance Analysis of Parallel Processing
PhD Thesis, Jan Lemeire
November 6, 2007

Goals of Performance Analysis

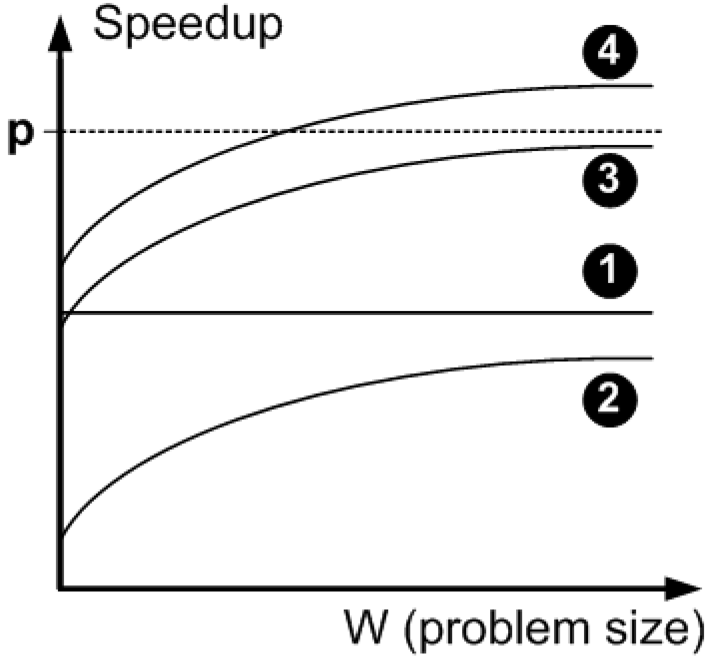
- Understanding of the computational process in terms of resource consumption
- Identification of inefficient patterns
- Performance prediction
- Performance characterization of program and system

Speedup vs. # of processors



1)Ideal, linear speedup
2)Increasing, sub-linear speedup
3)Speedup with an optimal number of processors
4)No speedup
5)Super-linear speedup

Speedup vs. problem size



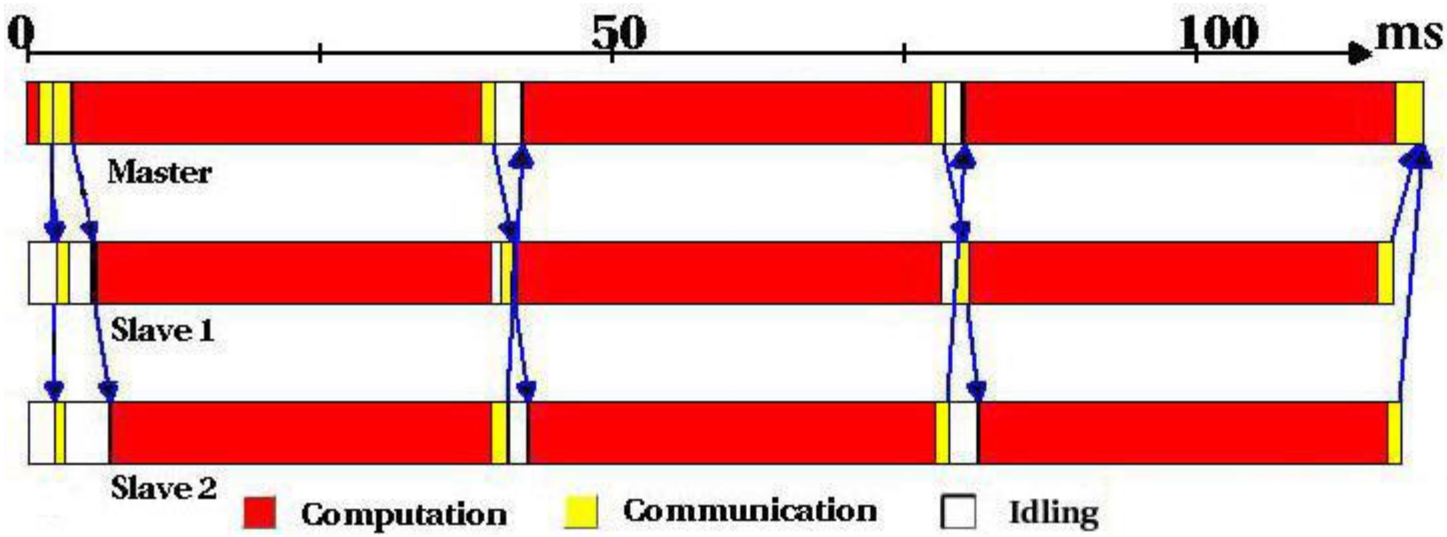
1)Constant speedup

2)Increasing, asymptotically, towards value sublinear speedup (< p)

3)Increasing towards p

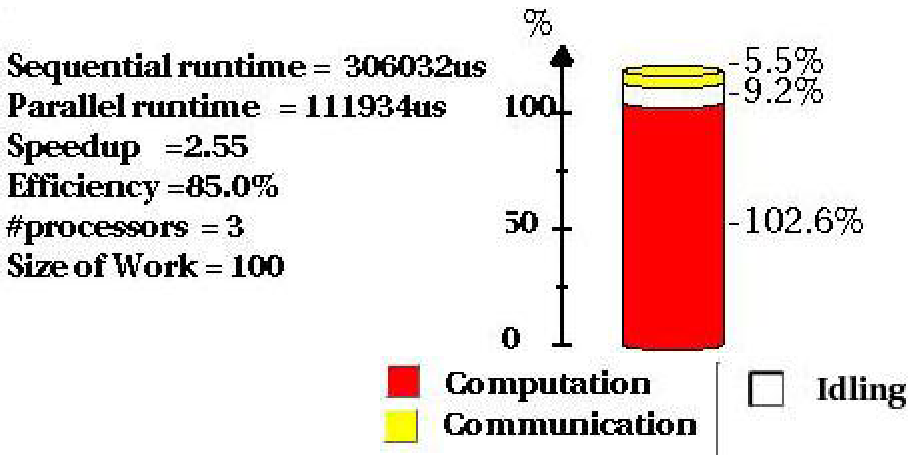
4)Increasing towards super-linear speedup

Parallel Matrix Multiplication



Speedup=2.55 Efficiency = 85%

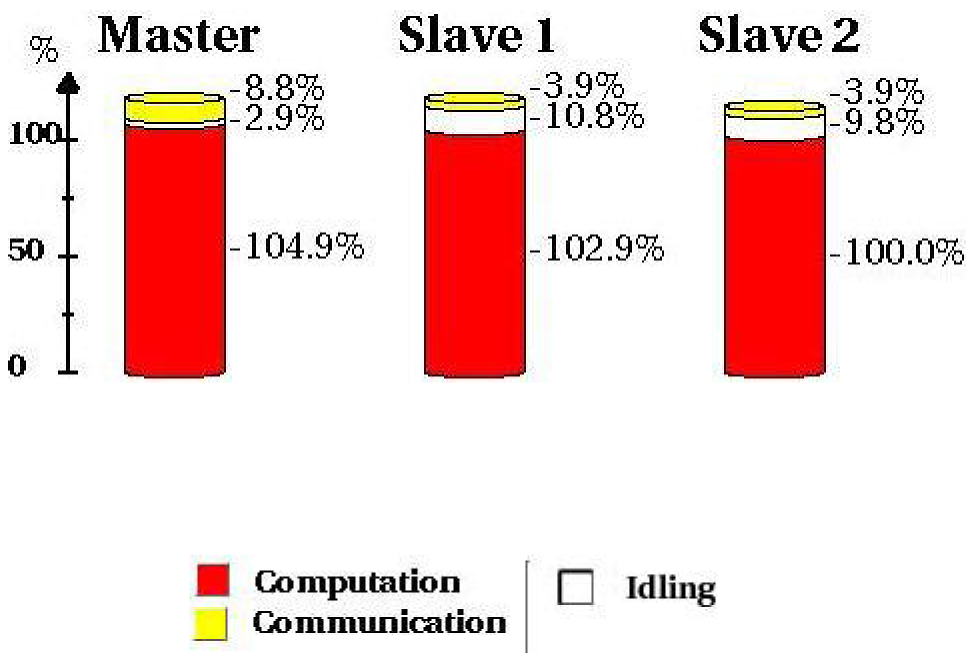
Parallel Matrix Multiplication



Parallel anomaly = 2.6%

- Overheads: the communication and the idle time.
- Their ratio with the sequential time is given.
- The sum of the processor's computation times divided by the sequential runtime is also given, but is not equal to 100%. A value of 100% means that the computation time of the useful work is equal for a sequential as for a parallel execution.
- It is 102.6% instead, which means that the overhead ratio of the parallel anomaly is 2.6%. In parallel, 2.6% more cycles are needed to do the same work.

Parallel Matrix Multiplication

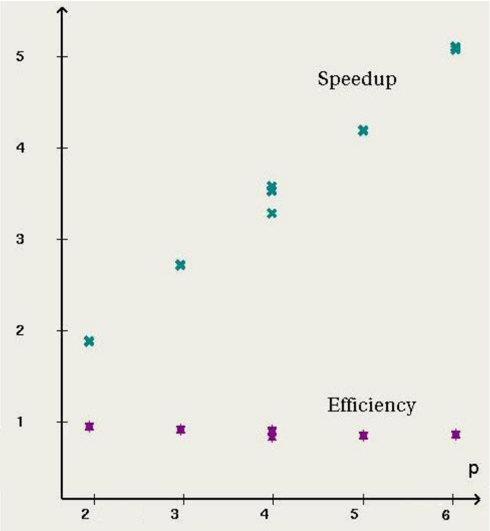


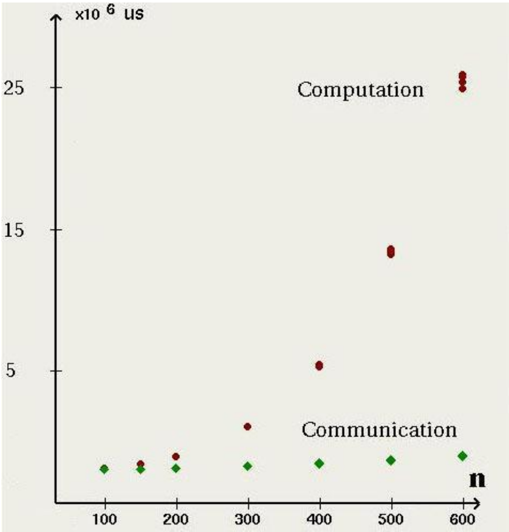
Overhead Classication

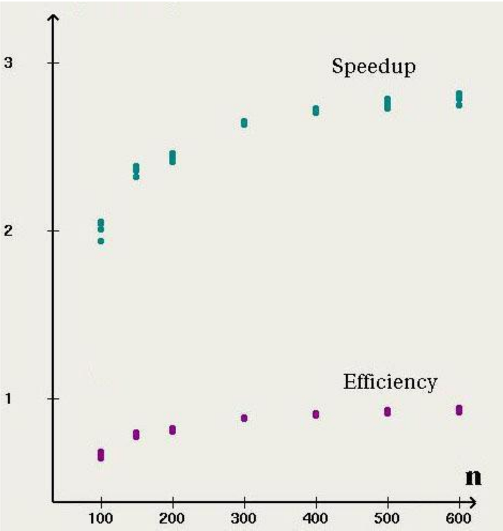
• Control of parallelism: extra functionality necessary for parallelization (like partitioning)
• Communication: overhead time not overlapping with computation
• Idling: processor has to wait for further information
• Parallel anomaly : useful work differs for sequential and parallel execution

$$T_{seq} + T_{anomaly} = \sum_{i}^{p} T_{work}^{i}$$

Overhead Classication

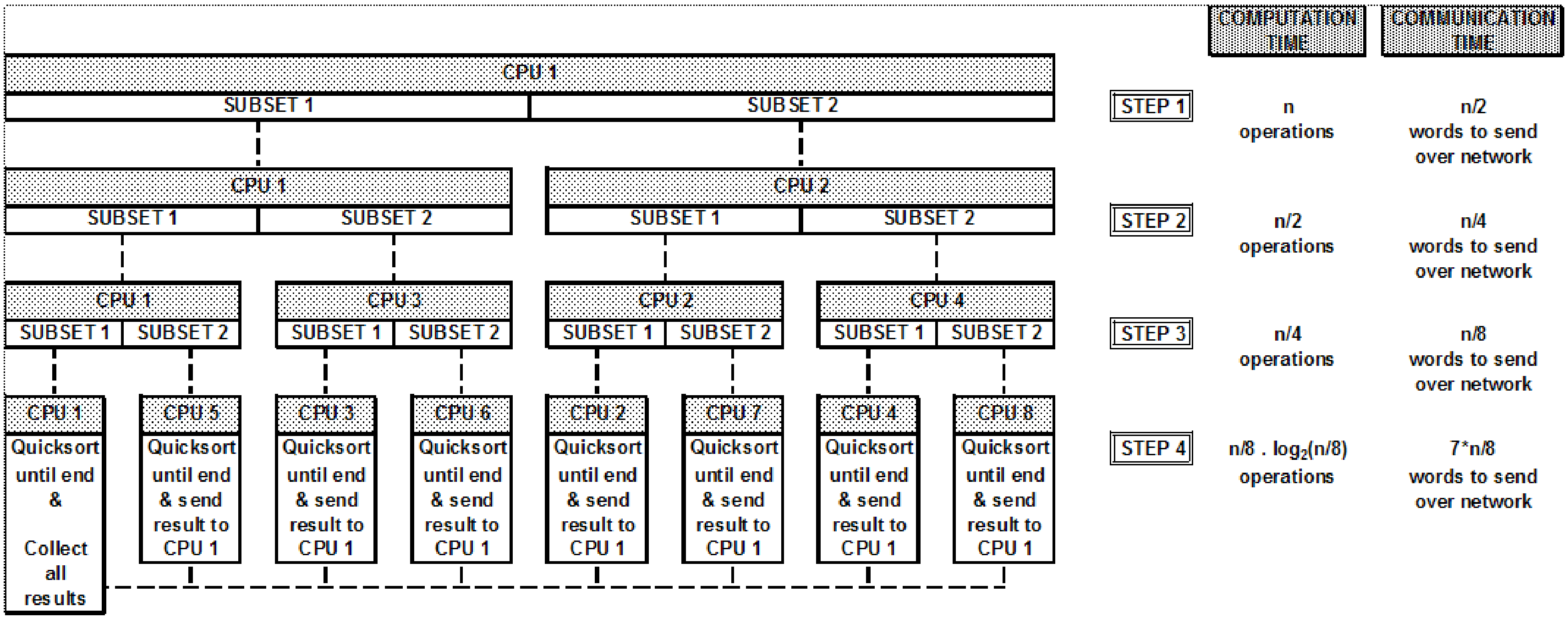






P: no. of processors
n: work size (matrix size)

Quicksort



COMPUTATION TIME
COMMUNICATION TIME
CPU 1
SUBSET 1
SUBSET 2
CPU 1
SUBSET 1
SUBSET 2
CPU 2
SUBSET 1
SUBSET 2
CPU 1
SUBSET 1
SUBSET 2
CPU 3
SUBSET 1
SUBSET 2
CPU 2
SUBSET 1
SUBSET 2
CPU 4
SUBSET 1
SUBSET 2
CPU 1
Quicksort until end & Collect all results
CPU 5
Quicksort until end & send result to CPU 1
CPU 3
Quicksort until end & send result to CPU 1
CPU 6
Quicksort until end & send result to CPU 1
CPU 2
Quicksort until end & send result to CPU 1
CPU 7
Quicksort until end & send result to CPU 1
CPU 4
Quicksort until end & send result to CPU 1
CPU 8
Quicksort until end & send result to CPU 1
STEP 1
n operations
n/2 words to send over network
STEP 2
n/2 operations
n/4 words to send over network
STEP 3
n/4 operations
n/8 words to send over network
STEP 4
n/8 . log2(n/8) operations
7*n/8 words to send over network

Overhead Optimization

1. Generate/draw execution profile
2. Identify lost cycles
3. Study impact on overhead
4. Determine causes of overhead
5. Plot performance in function of p and W