Analytical Modeling of Parallel Systems performance evaluation

Lecture 7 - MPP

References :

“Introduction to Parallel Computing” Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

“Distributed Systems: Paradigms and Models”, M.Danellutto

Why to evaluate the performance a-priori?

Evaluate the performance before starting writing application code!

- Performance orientation => In case several different implementations are possible, we are interested in knowing which one will provide the better performance.

- Productivity orientation => Which of the possible implementations will eventually provide performance that matching the performance requirements, and it is the less expensive in terms of coding effort.

Additionally:

- we may be interested in knowing how much *overhead* is added by a possible implementation.
- we may be interested in knowing what kind of performance we can achieve by using a given parallelism **exploitation pattern or skeleton** composition
  - in case expected performance of the given pattern doesn't match the requirements we can simply avoid to go on implementing a useless application.

Performance model

- The performance model measures the performance of
  an application on a given target machine
  in function of a full range of parameters depending either on:
  - the application features
    - parallelism exploitation
    - patterns used,
    - time spent in the execution of the sequential portions of code,
    - dimensions of the data types involved

  or

  – on the target machine features
    - communication times,
    - processing power of the different processing elements,
    - topology of the interconnection network,
    - features of the (shared) memory subsystem, etc.

Performance functions

=>a function P mapping values of some performance indicator.

More precisely we are interested in a set of functions $P_1; : : : ;P_h$

- each modeling a different performance indicator

- These performance model functions are used to:
  - predict performance of an application,
  - compare alternative implementations of an application, or
  - evaluate the overheads of an implementation of an application.

Level of accuracy

- Depending on the number of the parameters taken into account, as well as on the final “form" of the performance model, different results maybe achieved.

- different approximation levels =>
  - different accuracies may be achieved in the predicted results.

Topics Overview (covered in L6 & L7)

- Sources of Overhead in Parallel Programs
- Performance Measures/Metrics for Parallel Systems
- Effect of Granularity on Performance
- Asymptotic Analysis of Parallel Programs

- Scalability of Parallel Systems
- Minimum Execution Time and Minimum Cost-Optimal Execution Time
- Other Scalability Metrics

Analytical Modeling - Basics

- A sequential algorithm is evaluated by its runtime
  - in general, asymptotic runtime as a function of input size
- The asymptotic runtime of a sequential program is identical on any serial platform.
- The parallel runtime of a program depends on
  - the input size & problem complexity
  - the number of processors, and
  - the communication parameters of the machine.
  - ... memory latency...
- An algorithm must therefore be analyzed in the context of the underlying platform.
- A ***parallel system*** *inside an analytical model* is a combination of
  - a parallel algorithm and an
  - an underlying platform.

Sources of Overhead in Parallel Programs

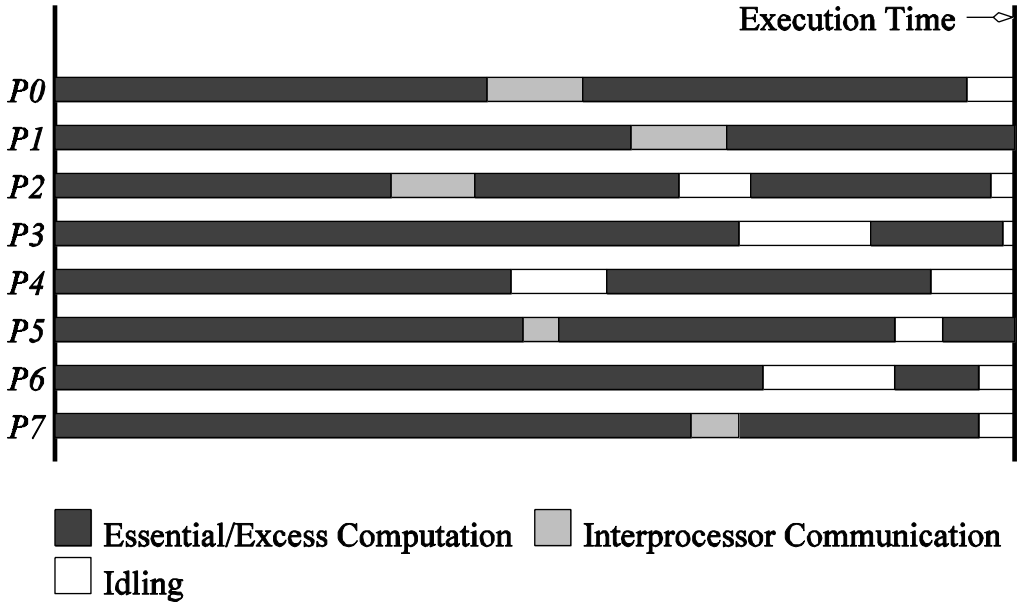
- If we use two processors, shouldn't our program run twice as fast?
  ☺
- No!!!
  ☹

A number of overheads, including:

- wasted/supplementary computation,
- communication,
- idling,
- contention

cause degradation in performance.

Example



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

- *Interprocess interactions*:
  Processes 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 the computation which is 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/synchronization.

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 $\boldsymbol{T_s}$ and the parallel runtime by $\boldsymbol{T_P}$.
- The same notation for time-complexity!

Analytical Modeling - Basics

A number of performance measures are intuitive:

- Wall clock time - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble.

  – But how does this scale when
    - the number of processors is changed

    or
    - the program is ported to another machine altogether?

  – How much faster is the parallel version? This implies the obvious follow-up question:
    – what's the baseline serial version with which we compare?

- Raw FLOPS count

  - What good are FLOPS counts? when they are not enough?

FLOPS = sockets x (cores/ sockets) x clock x (Flops/cycle)

Communication /Synchronization costs

- Distributed memory
  - The cost of communication is dependent on a variety of features including
    - the programming model semantics,
    - the network topology,
    - data handling and routing, and
    - associated software protocols.
- Shared memory
  - Synchronization

DISTRIBUTED MEMORY

Communication through messages

Communication time

- The total time to transfer a message over a network comprises of the following:
  - *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  - *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
  - *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

Store-and-Forward Routing

- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size *m* words to traverse *l* communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, t_h is small and the above expression can be approximated by

$$t_{comm} = t_s + mlt_w.$$

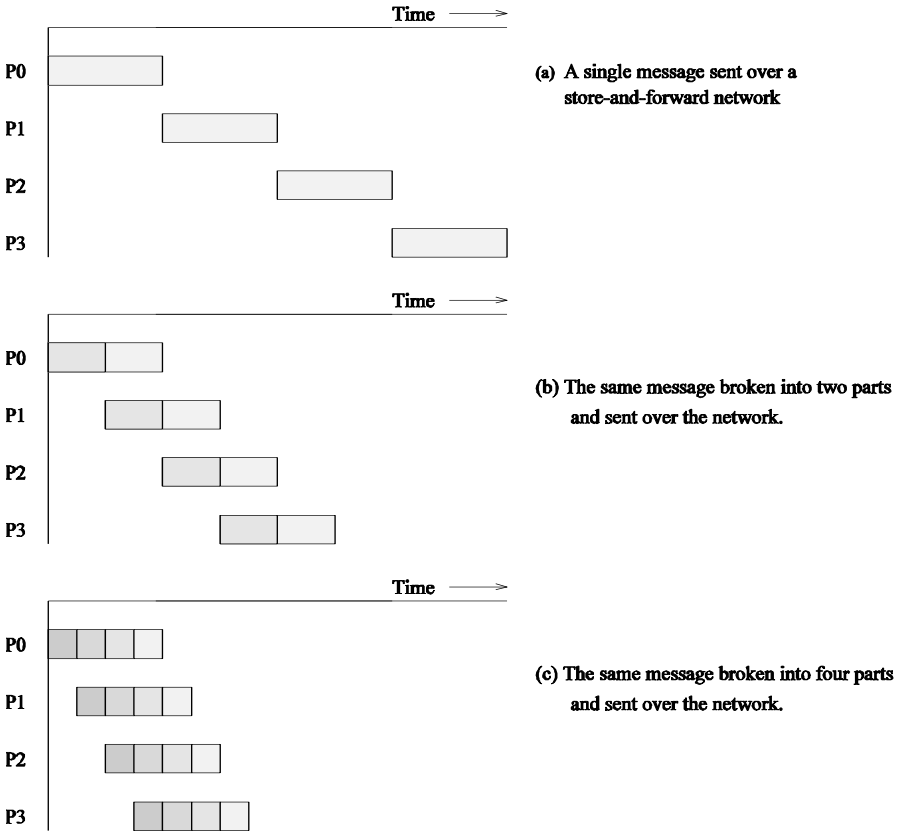
Packet Routing

- Store-and-forward makes poor use of communication resources.
- Packet routing breaks messages into packets and pipelines them through the network.
- Since ***packets may take different paths***, each packet must carry routing information, error checking, sequencing, and other related header information.
- The total communication time for packet routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

- The factor t_w accounts for overheads in packet headers.

Routing Techniques



Passing a message from node P_0 to P_3 (a) through a ***store-and-forward*** communication network; (b) and (c) extending the concept to ***cut-through routing***. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

Cut-Through Routing

- Takes the concept of packet routing to an extreme by further dividing messages into basic units called ***flits***.
- Since flits are typically small, the header information must be minimized.
- This is done by ***forcing all flits to take the same path***, in sequence.
- A tracer message first programs all intermediate routers.
  All flits then take the same route.
- ***Error checks are performed on the entire message***, as opposed to flits.
- No sequence numbers are needed.

Cut-Through Routing

- The total communication time for cut-through routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

- This is identical to packet routing, however, t_w is typically much smaller.

Simplified Cost Model for Communicating Messages

- The cost of communicating a message between two nodes away using cut-through routing is given by

$$t_{comm} = t_s + lt_h + t_w m.$$

- In this expression, t_h is typically smaller than t_s and t_w.

- For these reasons, we can approximate the cost of message transfer b

$$t_{comm} = t_s + t_w m.$$

Simplified Cost Model for Communicating Messages

- It is important to note that the original expression for communication time is valid for only uncongested networks.
- If a link takes multiple messages, the corresponding t_w term must be scaled up by the number of messages.
- Different communication patterns congest different networks to varying extents.
- It is important to understand and account for this in the communication time accordingly.

Cost Models for Shared Address Space Machines

- While the basic messaging cost applies to these machines as well, a number of other factors make accurate cost modeling more difficult.

- Memory layout is typically determined by the system.
- Spatial locality is difficult to model.

Caches:

- Finite cache sizes can result in cache thrashing.
- Cache hierarchies
- Overheads associated with invalidate and update operations are difficult to quantify.

- Prefetching can play a role in reducing the overhead associated with data access.

- False sharing and contention are difficult to model.

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.
- T_S is the serial time.
- $T_{all} - T_S$ is the total time spend by all processors in non-goal computation work. This is called the *overhead*.
- The total time collectively spent by all the processing elements could be approximate as:

  $T_{all} = p\ T_P$ (p is the number of processors).
- The overhead function (T_o) is therefore given by

$$T_o = p\ T_P - T_S$$

Performance Metrics for Parallel Systems: ***Speedup***

- What is the benefit from parallelism?

(general definition)

- Speedup (***S***) is the ratio of
  - the time taken to solve a problem on a single processor (the method used for solving could be different)

  to
  - the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

Alternative Definitions…

Speedup Factor

$$S(n) = \frac{\text{Execution time using one processor (single processor system)}}{\text{Execution time using a multiprocessor with n processors}} = \frac{t_s}{t_p}$$

where t_s is execution time on a single processor and t_p is execution time on a multiprocessor. $S(n)$ gives increase in speed by using multiprocessor. Underlying algorithm for parallel implementation might be (and is usually) different.

Speedup factor can also be cast in terms of computational steps:

$$S(n) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with n processors}}$$

Maximum speedup is (usually) n with n processors (*linear speedup*).

Speed-up types

- **relative**
  - T_s = parallel program executed on one processor

- **absolute**
  - consider the best sequential algorithm that solve the problem!

- Theoretical and empirical speedup

Amdahl's Law - pessimistic

- Amdahl's Law states the speedup of processing based on the percentages of the serial part and parallel part:

Considering that:
Total sequential execution time complexity is = 1
seq = sequential computation part(fraction);
par = parallel computation part(fraction)
1=seq+par
n= # processors

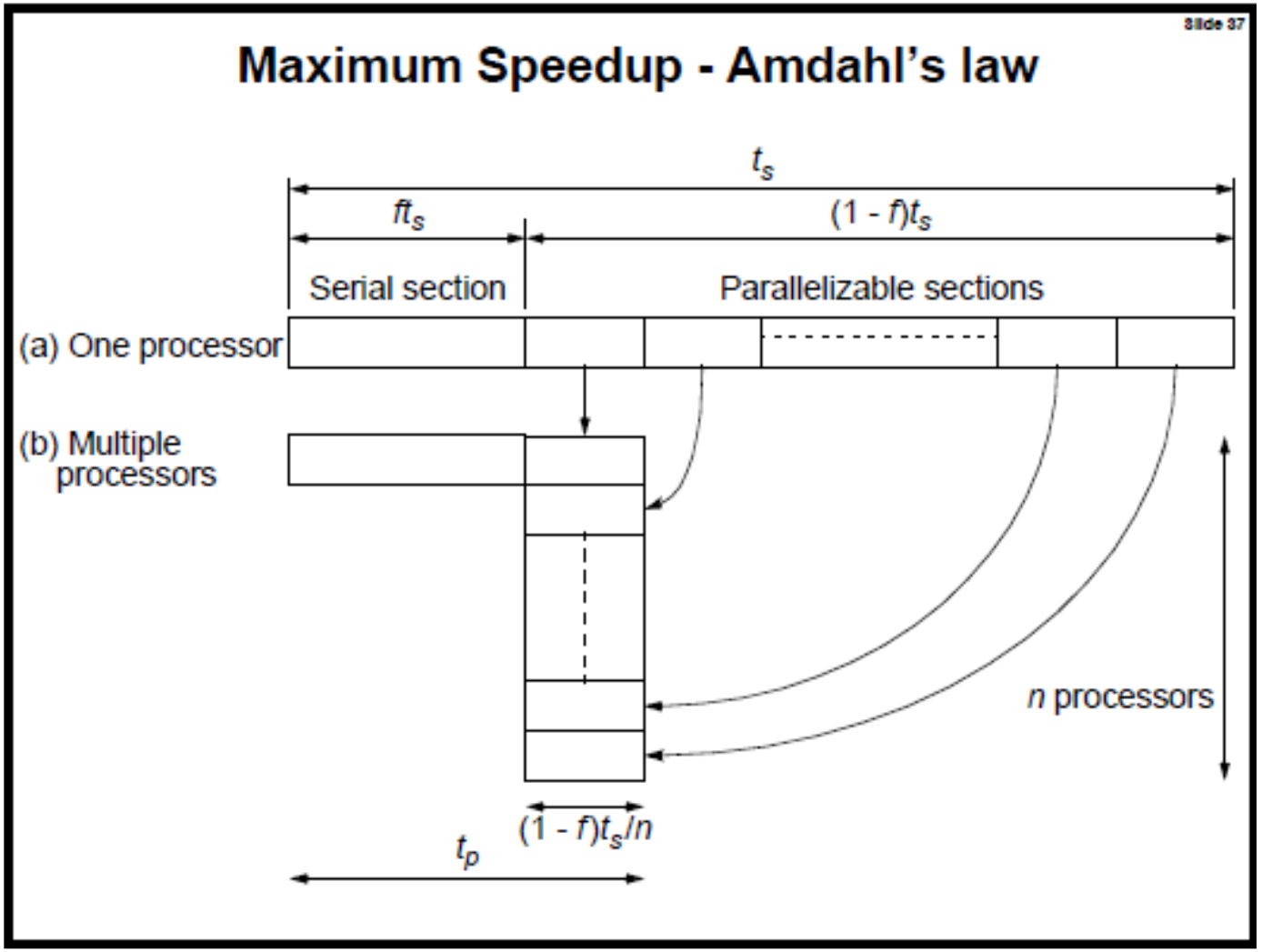
speedup = 1/(seq + par/n)

- When *n* -> *infinity*, then the speedup approaches to 1/seq.
- That is to say the upper limit of speedup is bounded by the cost of the serial part.
- But Amdahl's Law does not take into account the size of the problem being solved!

Alternative presentation

$$S(n) = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1+(n-1)f}$$

seq= f ;
par = 1-f



Amdahl's law

$S(n) = (f + (1-f)/n)^{-1}$

$$S(n) \leq f^{(-1)}$$

- If f=5% then $S(n) \leq 100/5=20$ even if n=10000…, n=1000…
- The speed-up of a program from parallelization is limited by how much of the program can be parallelized.
- This is true iff **f** does not depend on the size of the problem!

Gustafson's law - optimistic

- Gustafson's law is another law in computing, closely related to Amdahl's law. It states that the speedup with *n* processors is

Remarks:

- Amdahl's law assumes:
  - a fixed problem size and that
  - the running time of the sequential part (fraction) of the program is independent of the number of processors,
  - the fraction of the sequential part remains the same independent on the problem size
- Gustafson's law takes the opposite view
- Gustafson's Law rationalizes that
  - as the size of the problem grows, the serial part will become a smaller and smaller percentage of the entire process

Gustafson's law – more detailed analysis

- Let m = size of the problem, n = # of processors,

Gustafson's Law states that:

considering that:

T_p = seq(m) + par(m) = 1 par(m) = 1 – seq(m),

when m -> infinity, seq(m) -> 0,

then

speedup =seq(m) + n(1 – seq(m))

when n processor are applied for the parallel part

- As m -> infinity, and as seq(m) becomes a smaller and smaller percentage, the speedup approaches n.
- In other words, as programs get larger, having multiple processors will become more advantageous, and it will get close to n times the performance with n processors as the percentage of the serial part diminishes.
- **It assumes the absolute cost of the serial part is constant and does not grow with the size of the problem.**

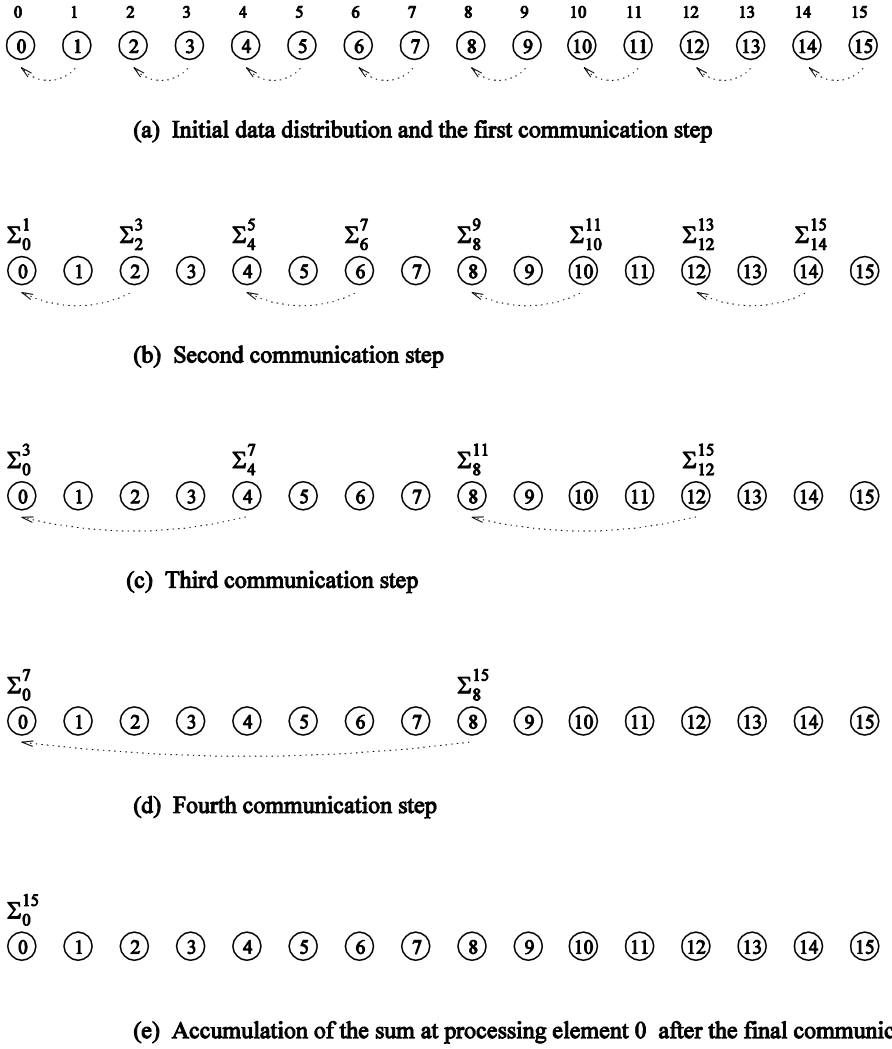
Order of magnitude - Example

- Assume that a task has two independent parts, A and B.
- B takes roughly 25% of the time of the whole computation.
- With effort, a programmer may be able to make this part (B) five times faster, but this only reduces the time for the whole computation by a little.
- In contrast, one may need to perform less work to make part A twice as fast.
- This will make the computation much faster than by optimizing part B, even though B got a greater speed-up (5 × versus 2 ×).

Performance Metrics: Example

- Consider the problem of adding ***n*** numbers by using ***n*** processing elements.
- If ***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 processors.

Performance Metrics: Example



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

Σ_i^j denotes the sum of numbers with consecutive labels from ***i*** to ***j***.

Performance Metrics evaluation:
Adding on a distributed memory systems

=>If an addition takes constant time - t_c and

- communication of a single word takes time $t_s + t_w$,
- $p=n$

$$T_P = \log n \, [(t_s + t_w) + t_c]$$

$$T_S = n \, t_c$$

$$S(n) = T_S / T_P = (n / \log n) \, (t_c / (t_s + t_w + t_c))$$

$$T_P = \Theta (\log n)$$

- We know that $T_S = \Theta (n)$
- Asymptotic Speedup S is given by $S = \Theta (n / \log n)$

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, usually the best sequential program is considered as the baseline.

Performance Metrics: Speedup Example – relative versus absolute -

- Consider the problem of parallel bubble sort.
- The serial time for bubblesort 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. (relative speed-up!)
- But is this really a fair assessment of the system?
- What if serial quicksort only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the parallel system.

Performance Metrics: Speedup Bounds

- Speedup can be as low as 0 (the parallel program never terminates).
- Speedup, in theory, should be upper bounded by n *(n = #processors)* – we can only expect a n-fold speedup if we use as many resources we need.
  - A speedup greater than n is possible only if each processing element spends less than time T_S / n 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.

Asymptote analysis

- The speedup maximum asymptote is given by the linear function- f(n) = n.

(an asymptote is a **line** that a curve approaches, as it heads towards infinity)

Proof:

- If we assume that $s(n) > n$ then the portions of the algorithm executed in parallel on the n processing elements will take a time
  $Tp < Tseq/n$ and therefore we could have performed sequentially the n computations taking Tp on a single processing element, taking an overall time

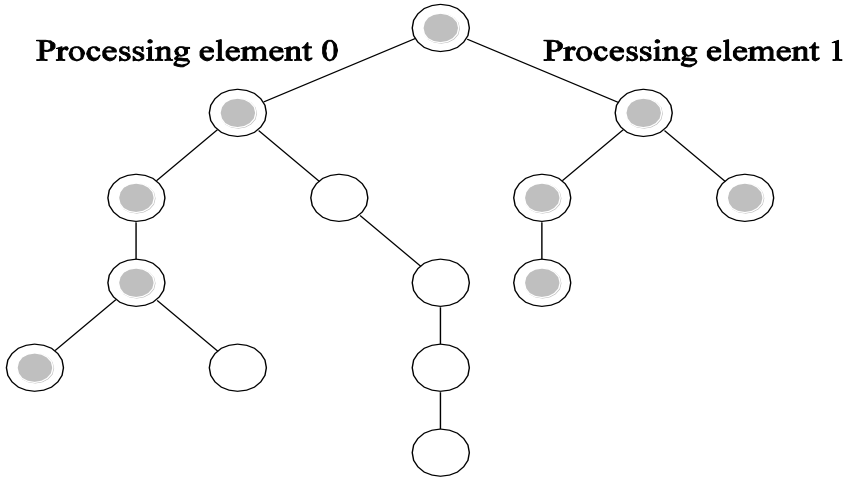
$T_{all} = n*Tp < n*Tseq / n < Tseq$

which is in contradiction with the assumption Tseq was the “best" sequential time.

- But there are exceptions…

Performance Metrics: Superlinear Speedups

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



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.

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. If ***two*** processors are used, since the problem size/processor is smaller (less data), 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, this corresponds to a speedup of 2.43!

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

$$E = \frac{S}{p}. \quad (2)$$

- Following the bounds on speedup, efficiency can be as low as 0 and as high as 1.

Cost of a Parallel System

- Cost is the product of parallel runtime and the number of processing elements used $C = p \times T_P$
- 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 - $C = \Theta(T_s)$
- Since $E = T_S / p\,T_P$, for cost optimal systems, $E = O(1)$.
- Cost is sometimes referred to as ***Work*** or *processor-time product*.

Impact of Non-Cost-Optimality

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 $p\ T_P$ product of this algorithm is $n\ (\log n)^2$.

- This algorithm is not cost optimal but only by a factor of $\log n$.

- If $p < n$, assigning the n tasks to p processors gives $T_P = (\log n)^2 (n / p)$.

- The corresponding speedup of this formulation is $p / \log n$.

- For a given **p** this speedup goes down as the problem size n is increased!