THE UNIVERSITY OF WARWICK

High Performance Computing

Course Notes

Performance I

Dr Ligang He

Metrics to measure the parallelization quality of parallel programs

Degree of Parallelism, average parallelism

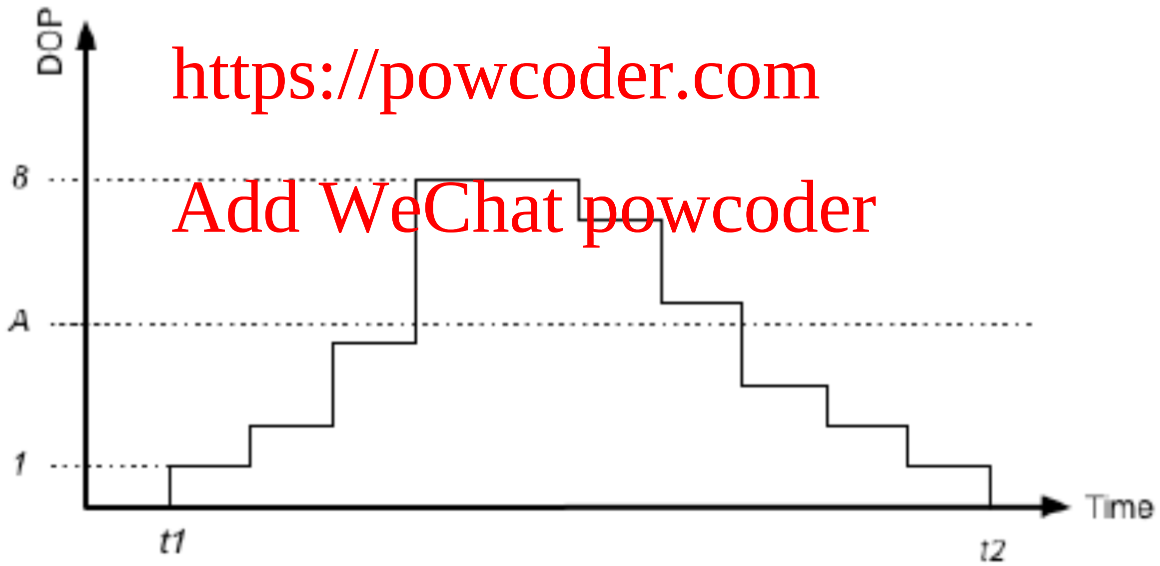
Effective work

Speedup

Parallel efficiency

Degree of Parallelism

- **Degree of Parallelism (DOP)**
  - **The number of processors engaged in execution at the same time**
  - **Two forms of functions: continuous form and discreet form**



Degrees of Parallelism

Factors that affect the DOP include:

- Application properties
  - Data dependency,
  - Communication overhead
- Resource limitations
  - number of processors,
  - memory, I/O
- Algorithms
  - how does the algorithm divide up work?

Effective Work

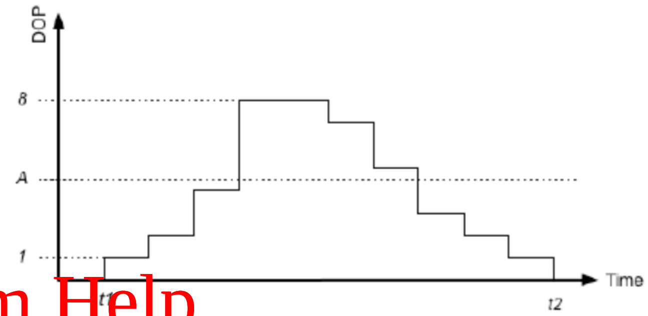
Effective Work

- This is the total amount of computation executed within a given time interval.

- Effective work relates to DOP

Effective work

Calculating Effective work



- ***m* homogeneous processors**
- **processing capacity of a single processor (execution rate) = Δ**
- ***DOP(t)*=Number of busy PEs at time t in [t1, t2]**

- **Total effective work in discrete form**

$$W = \Delta \sum_{i=1}^{m} i.t_i$$

where t_i is the total time that DOP $= i$ and $\sum_{i=1}^{m} t_i = t_2 - t_1$

- **Total effective work in continuous form:**

$$W = \Delta \int_{t_1}^{t_2} \mathrm{DOP(t)}\,dt$$

Average Parallelism

Average parallelism:

- **Continuous form:**

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \mathrm{DOP}(t)\,dt$$

- **Discrete form:**

$$A = \frac{\sum_{t=1}^{m} i.t_i}{\sum_{t=1}^{m} t_i}$$

Speedup

We desire to know the improvement (or not) brought about by parallelising an application code.

The improvement can be measured by *speedup*

- In the simplest form, speedup is the ratio of execution time of a serial implementation to the execution time of a parallel implementation.

If *n* processors are used, then:

$$S(n) = t_1/t_n$$

- t_1 is the **worst case** execution time of the **optimal** serial implementation.
- t_n is the **worst case** execution time of the parallel algorithm using *n* processors.

Speedup

What is "good" speedup?

- Linear speedup is regarded as optimal
- Maximum speedup for a parallel algorithm with *n* processors is *n.*
- To illustrate this:
  - Consider the execution time of an application is t_1
  - The application is split into *n* processes
  - Assume no overheads, communications, synchronisation etc.
  - The least execution time is t_1 / n
  - So the maximum speedup is $S(n) = t_1 / (t_1 / n) = n$
- Not always true (we may achieve superlinearity in some special circumstances)

Speedup

Some tasks can exceed linear speedup.

- **This is superlinear speedup (S*(n)* > *n*)**

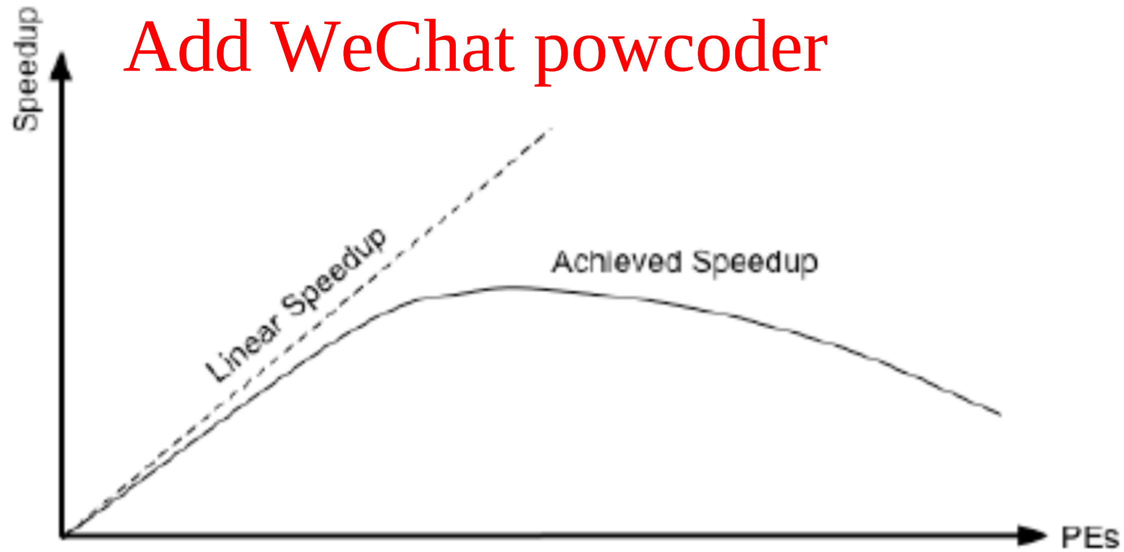
Reasons

- **Cache or memory effects**

- **Evidence of sub-optimal sequential implementation**

Speedup

- The general trend of speedup as the number of processors increases
    - First, speedup increases as the number of PE increases, but the gap with the maximum speed also increases
    - After speedup reaches a maximum speedup, adding processors further is of no benefit and will harm performance.



Speedup

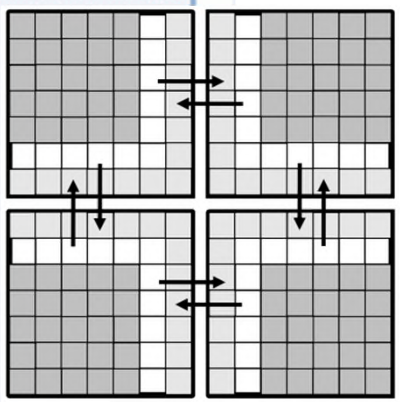
How to obtain good *S(n)* for large *n*?

- Algorithm design: Minimal sequential component and good percentage of inherent (data) parallelism.

- workload management: balancing workload among processors

- When there are different ways to partition data and achieve load balance, try to maintain a high ratio of computation to communication (computation represents effective work while communication represents overhead)

  - Use the way which leads to less communication
  - Low frequency of communications between processors
  - increase the size of the work run by each processor.



Reducing the Impact of Communication

Communication has crucial impact on the performance of parallel programming

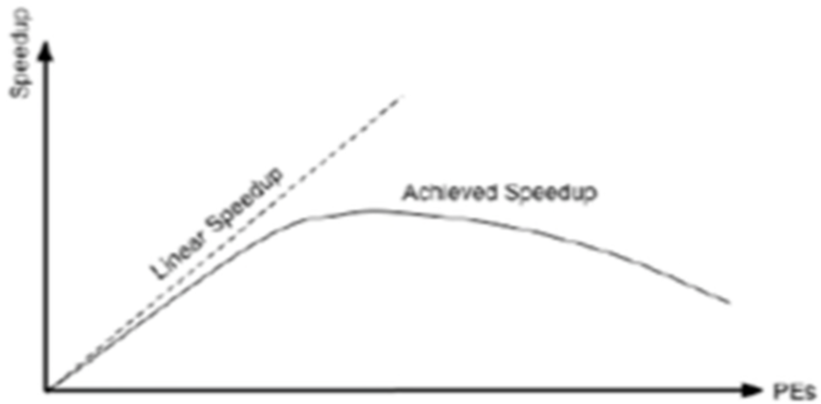
How to reduce the impact of communication:

- Minimize the amount of communication (e.g. by maintaining a good data locality)

- Overlap communications with computation where possible.

- Reduce latency and overhead by sending a few large messages, rather than a lot of small messages.
- At the hardware level, can reduce latency by using fast (but expensive) communications.

Parallel efficiency



Parallel efficiency:

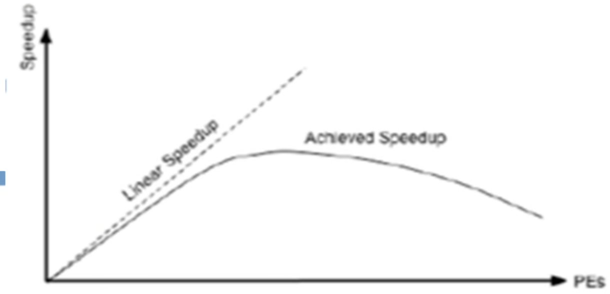
$$E(n) = S(n) / n$$

Parallel programs are not usually 100% efficient, i.e. S(n) << n

Main issues that affect parallel efficiency are:

- Same as the factors for affecting speedup
- Plus the impact of n; typically, greater n, lower efficiency

Iso-efficiency



Constant Efficiency:

- How the amount of computation performed (N) must scale with processor number P to keep parallel efficiency E constant
- The function of N over P is called an algorithm's iso-efficiency function
- An algorithm with an iso-efficiency function of O(P) is highly scalable
  - E.g., Increase p by three times, only need to increase N by three times to maintain efficiency
- An algorithm with a quadratic or exponential iso-efficiency function is less scalable
  - E.g. increase p by three times, need to increase N by 9 times and 8 times, respectively

Work out the iso-efficiency function

- Given the parallel efficiency function as follows, work out the iso-efficiency function.

$$E_2 = \frac{5N^2}{5N^2 + 2P^2 + 4NP}$$

Answer: N=O(P)

Four approaches to modelling the performance of a parallel application

Speedup

Amdahl's law

Asymptotic analysis

Modelling execution time

Speedup approach

Using speedup approach, we can say something like “this algorithm achieved a speedup of S on p processors with problem size *N*”

This approach can give us some ideas about the algorithm quality, but we cannot judge the quality of an algorithm by a single speedup data

Elaborate this point in the following example

Speedup approach

Consider a sequential algorithm and its optimal execution time $T = N + N^2$, where N is the problem size

- Parallel Algorithm 1: $T = N + (N^2 / p)$
  - Partitions the computationally expensive $O(N^2)$
  - No other costs.
- Parallel Algorithm 2: $T = ((N + N^2) / p) + 100$
  - Partitions the whole computation
  - Introduces fixed overhead cost of 100.
- Parallel Algorithm 3: $T = ((N + N^2) / p) + 0.6p^2$
  - Partitions the whole computation
  - Introduces variable overhead cost of $0.6p^2$

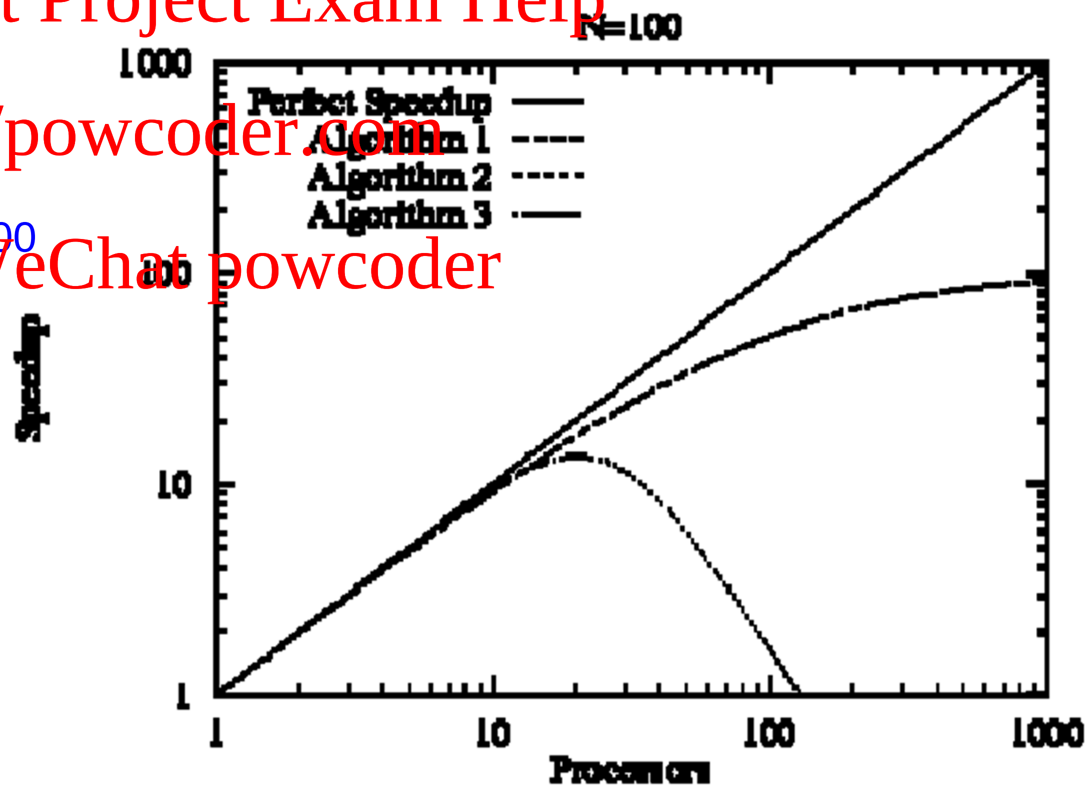
Speedup approach

These algorithms all achieve a speedup of about 10.8 when p = 12 and N = 100 , but differentiates with each other when p becomes large.

Algorithm 1: $T = N + (N^2 / p)$

Algorithm 2: $T = ((N + N^2) / p) + 100$

Algorithm 3: $T = ((N + N^2) / p) + 0.6p^2$



Amdahl's Law

Applications may contain elements that are not amenable to parallelisation.

Let this serial fraction be f:

- If we make the remaining part n times faster by running it on n processors, then the time T_n is:

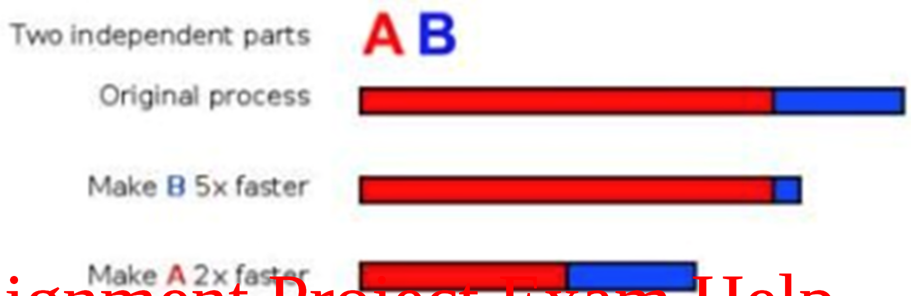
$$T_n = \frac{(1-f)T_1}{n} + fT_1$$

- Hence, speedup is: $S(n) = \dfrac{n}{(1-f)+nf} \leq \dfrac{1}{f}$

- For example, an application does a final (non-parallelisable) collective operation at the end of each iteration which accounts for 8% of the computation time - the maximum achievable speedup is 12.5.

- This is Amdahl's Law.

Application of Amdahl's Law

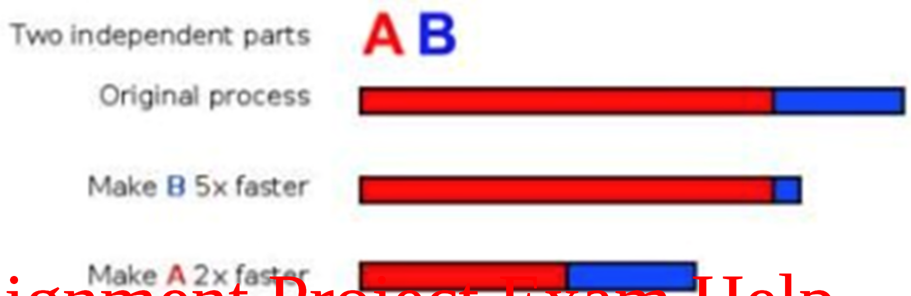


- Part A takes 75% and part B takes 25% of the whole computation time

- If we decide to parallelize part B, then the upper bound of the speedup is 1/0.75=1.33

- If we decide to parallelize part A, then the upper bound of the speedup is 1/0.25=4

Application of Amdahl's Law



- Part A takes 75% and part B takes 25% of the whole computation time

- If we make part B 5 times faster, then

$$\text{speedup} = \frac{1}{\frac{0.25}{5} + 0.75} = 1.25$$

- If we make part A 2 times faster, then

$$\text{speedup} = \frac{1}{0.25 + \frac{0.75}{2}} = 1.6$$

- Therefore, making A twice faster is better (and typically be much easier) than making B five times faster;

Amdahl's Law

- **Amdahl's law shows us the limitation of parallelising codes**
- **Disadvantages**

  - Can only tell the upper bound of the speedup for a particular algorithm

  - Cannot tell if other algorithms with greater parallelism exist for the problem.

Asymptotic analysis

- In this modelling approach, we can say something like "the algorithm takes the time of O(nlogn) on n processors"

- Disadvantage:

  - ignore the lower-order term:

    - e.g. given an algorithm with time complexity of O(nlogn), the actual time complexity could be 10n+nlogn, when n is small, 10n dominates

  - Only tell the order of the execution time of a program, not its actual execution time:

    - e.g. given two algorithms, one's time complexity is 1000nlogn while the other's is $10n^2$. 1000nlogn is better than $10n^2$ when n exceeds a certain value, but $10n^2$ is less than 1000nlogn when n is less than the value