1 Performance Analysis

- Analysis of the performance measures of parallel programs.
- Two computational models are studied, namely *the equal duration* processes and the *parallel computation with serial sections* models.
- Two measures; the speedup factor and the efficiency.
- The impact of the communication overhead on the overall speed performance of multiprocessors is emphasized in these models.
- Discussion on the scalability of parallel systems.

1.1 Computational Models

In developing a computational model for multiprocessors, we assume that a given computation can be divided into concurrent tasks for execution on the multiprocessor.

1.1.1 Equal Duration Model

- In this model, it is assumed that a given task can be divided into n equal subtasks, each of which can be executed by one processor.
- If t_s is the execution time of the whole task using a single processor, then the time taken by each processor to execute its subtask is $t_p = t_s/n$.
- Since, according to this model, all processors are executing their subtasks simultaneously, then the time taken to execute the whole task is $t_p = t_s/n$.
- The speedup factor of a parallel system can be defined as the ratio between the time taken by a single processor to solve a given problem instance to the time taken by a parallel system consisting of n processors to solve the same problem instance.

$$S(n) = \frac{t_s}{t_p} = \frac{t_s}{t_s/n} = n \tag{1}$$

- The above equation indicates that, according to the equal duration model, the speedup factor resulting from using n processors is equal to the number of processors used, n.

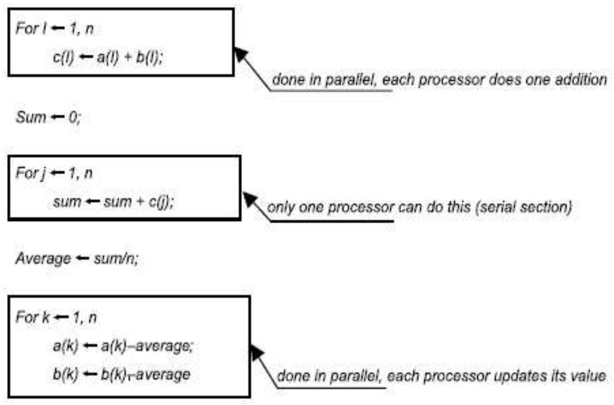
- One important factor has been overlooked in the above derivation. This factor is the communication overhead, which results from the time needed for processors to communicate and possibly exchange data while executing their subtasks.
- Assume that the time taken due to the communication overhead is called t_c then the actual time taken by each processor to execute its subtask is given by

$$S(n) = \frac{t_s}{t_p} = \frac{t_s}{t_s/n + t_c} = \frac{n}{1 + n * t_c/t_s} \tag{2}$$

- The above equation indicates that the relative values of t_s and t_c affect the achieved speedup factor.
- A number of cases can then be studied:
  1. if $t_c \ll t_s$ then the potential speedup factor is approximately n
  2. if $t_c \gg t_s$ then the potential speedup factor is $t_s/t_c \ll 1$
  3. if $t_c = t_s$ then the potential speedup factor is $n/n + 1 \cong 1$, for $n \gg 1$.
- In order to scale the speedup factor to a value between 0 and 1, we divide it by the number of processors, n. The resulting measure is called the efficiency, E.
- The efficiency is a measure of the speedup achieved per processor. According to the simple equal duration model, the efficiency E is equal to 1 if the communication overhead is ignored.
- However if the communication overhead is taken into consideration, the efficiency can be expressed as

$$E = \frac{1}{1 + n * t_c/t_s} \tag{3}$$

- Although simple, the equal duration model is however unrealistic. This is because it is based on the assumption that a given task can be divided into a number of equal subtasks that can be executed by a number of processors in parallel.
- However, it is sufficient here to indicate that real algorithms contain some (serial) parts that cannot be divided among processors. These (serial) parts must be executed on a single processor.



```
For I ← 1, n
    c(I) ← a(I) + b(I);

Sum ← 0;

For j ← 1, n
    sum ← sum + c(j);

Average ← sum/n;

For k ← 1, n
    a(k) ← a(k)–average;
    b(k) ← b(k)–average
```

done in parallel, each processor does one addition
only one processor can do this (serial section)
done in parallel, each processor updates its value

Figure 1: Example program segments.

- Consider, for example, the program segments given in Fig. 1. In these program segments, we assume that we start with a value from each of the two arrays (vectors) a and b stored in a processor of the available n processors.
  - The first program block can be done in parallel; that is, each processor can compute an element from the array (vector) c. The elements of array c are now distributed among processors, and each processor has an element.
  - The next program segment cannot be executed in parallel. This block will require that the elements of array c be communicated to one processor and are added up there.
  - The last program segment can be done in parallel. Each processor can update its elements of a and b.

1.1.2 Parallel Computation with Serial Sections Model

- In this computational model, it is assumed that a fraction f of the given task (computation) is not dividable into concurrent subtasks. The remaining part $(1 - f)$ is assumed to be dividable into concurrent subtasks.

- The time required to execute the task on n processors is $t_p = t_s * f + (1 - f) * (t_s/n)$. The speedup factor is therefore given by

$$S(n) = \frac{t_s}{t_s * f + (1 - f) * (t_s/n)} = \frac{n}{1 + (n - 1) * f} \tag{4}$$

- According to this equation, the potential speedup due to the use of n processors is determined primarily by the fraction of code that cannot be divided.

- If the task (program) is completely serial, that is, $f = 1$, then no speedup can be achieved regardless of the number of processors used.

- This principle is known as Amdahl's law. It is interesting to note that according to this law, the maximum speedup factor is given by

$$lim_{n \to \infty} S(n) = \frac{1}{f}$$

- Therefore, according to Amdahl's law the improvement in performance (speed) of a parallel algorithm over a sequential one is limited not by the number of processors employed but rather by the fraction of the algorithm that cannot be parallelized.

- According to Amdahl's law, researchers were led to believe that a substantial increase in speedup factor would not be possible by using parallel architectures.

- The effect of the communication overhead on the speedup factor, given that a fraction, f, of the computation is not parallelizable.

$$S(n) = \frac{t_s}{t_s * f + (1 - f) * (t_s/n) + t_c} = \frac{n}{(n-1) * f + 1 + n * (t_c/t_s)} \tag{5}$$

  The maximum speedup factor under such conditions is given by

$$lim_{n \to \infty} S(n) = lim_{n \to \infty} \frac{n}{(n-1) * f + 1 + n * (t_c/t_s)} = \frac{1}{f + (t_c/t_s)}$$

- The above formula indicates that the maximum speedup factor is determined not by the number of parallel processors employed but by the fraction of the computation that is not parallelized and the communication overhead.

- Recall that the efficiency is defined as the ratio between the speedup factor and the number of processors, n. The efficiency can be computed as:

$$\begin{aligned} E(no\ communication\ overhead) &= \tfrac{1}{1+(n-1)*f} \\ E(with\ communication\ overhead) &= \tfrac{1}{(n-1)*f+1+n*(t_c/t_s)} \end{aligned} \tag{6}$$

- As the number of processors increases, it may become difficult to use those processors efficiently. In order to maintain a certain level of processor efficiency, there should exist a relationship between the fraction of serial computation, f, and the number of processor employed.

1.2 Skeptic Postulates For Parallel Architectures

A number of postulates were introduced by some well-known computer architects expressing about the usefulness of parallel architectures.

1.2.1 Grosch's Law

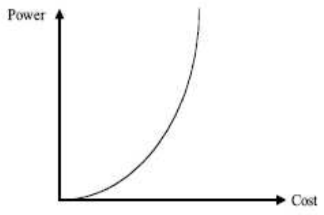


Figure 2: Power-cost relationship according to Grosch's law.

- It was as early as the late 1940s that H. Grosch studied the relationship between the power of a computer system, P, and its cost, C.
- He postulated that $P = K * C^s$, where s and K are positive constants. Grosch postulated further that the value of s would be close to 2. Simply stated, Grosch's law implies that the power of a computer system increases in proportion to the square of its cost (see Fig. 2).
- Alternatively, one can express the cost of a system as $C = sqrt(P/K)$ assuming that $s = 2$.
- According to Grosch's law, in order to sell a computer for twice as much, it must be four times as fast. Alternatively, to do a computation twice as cheaply, one has to do it four times slower. With the advances in computing, it is easy to see that Grosch's law is overturned, and it is possible to build faster and less expensive computers over time.

1.2.2 Amdahl's Law

- We defined the speedup factor of a parallel system as the ratio between the time taken by a single processor to solve a given problem instance to the time taken by a parallel system consisting of n processors to solve the same problem instance (see Eqn. 4).

- Similar to Grosch's law, Amdahl's law made it so pessimistic to build parallel computer systems due to the intrinsic limit set on the performance improvement (speed) regardless of the number of processors used.

- An interesting observation to make here is that according to Amdahl's law, f is fixed and does not scale with the problem size, n. However, it has been practically observed that some real parallel algorithms have a fraction that is a function of n. Let us assume that f is a function of n such that $lim_{n \to \infty} f(n) = 0$

$$lim_{n \to \infty} S(n) = lim_{n \to \infty} \frac{n}{1 + (n-1) * f(n)} = n \tag{7}$$

- This is clearly in contradiction to Amdahl's law. It is therefore possible to achieve a linear speedup factor for large-sized problems, given that $lim_{n \to \infty} f(n) = 0$, a condition that has been practically observed.

- For example, researchers at the Sandia National Laboratories have shown that using a 1024-processor hypercube multiprocessor system for a number of engineering problems, a linear speedup factor can be achieved.

- Consider, for example, the well-known engineering problem of multiplying a large square matrix $A(m \times m)$ by a vector $X(m)$ to obtain a vector, that is, $C(m)$. Assume further that the solution of such a problem is performed on a binary tree architecture consisting of n nodes (processors).
  - Initially, the root node stores the vector $X(m)$ and the matrix $A(m \times m)$ is distributed row-wise among the n processors such that the maximum number of rows in any processor is $m/n + 1$.
  - A simple algorithm to perform such computation consists of the following three steps:
    1. The root node sends the vector $X(m)$ to all processors in $O(m * logn)$

2. All processors perform the product $C_i = \sum_{j=1}^{m} a_{ij} * x_j$ in

$$O(m * (m/n + 1)) = O(m) + O(\frac{m^2}{n})$$

3. All processors send their C_i values to the root node in $O(m * logn)$.

– According to the above algorithm, the amount of computation needed is

$$O(m * logn) + O(m) + O(\frac{m^2}{n}) + O(m * logn) = O(m^2)$$

– The indivisible part of the computation is equal to

$$O(m) + O(m * logn)$$

– Therefore, the fraction of computation that is indivisible

$$f(m) = \frac{(O(m) + O(m * logn))}{O(m^2)} = O(\frac{(1 + logn)}{m})$$

– Notice that $lim_{m \to \infty} f(m) = 0$. Hence, contrary to Amdahl's law, a linear speedup can be achieved for such a large-sized problem.

- It should be noted that in presenting the above scenario for solving the matrix vector multiplication problem, we have assumed that the memory size of each processor is large enough to store the maximum number of rows expected.

- This assumption amounts to us saying that with n processors, the memory is n times larger. Naturally, this argument is more applicable to message passing parallel architectures than it is to shared memory ones.

1.2.3 Gustafson-Barsis's Law

- In 1988, Gustafson and Barsis at Sandia Laboratories studied the paradox created by Amdahl's law and the fact that parallel architectures comprised of hundreds of processors were built with substantial improvement in performance.

- In introducing their law, Gustafson recognized that the fraction of indivisible tasks in a given algorithm might not be known a priori. They argued that in practice, the problem size scales with the number of processors, n. This contradicts the basis of Amdahl's law.

- Recall that Amdahl's law assumes that the amount of time spent on the parts of the program that can be done in parallel, $(1-f)$, is independent of the number of processors, n.

- Gustafson and Brasis postulated that when using a more powerful processor, the problem tends to make use of the increased resources. They found that to a first approximation the parallel part of the program, not the serial part, scales up with the problem size.

- They postulated that if s and p represent respectively the serial and the parallel time spent on a parallel system, then $s + p * n$ represents the time needed by a serial processor to perform the computation.

- They therefore, introduced a new factor, called the scaled speedup factor, $SS(n)$, which can be computed as:

$$SS(n) = \frac{s + p * n}{s + p} = s + p * n = s + (1 - s) * n = n + (1 - n) * s \quad (8)$$

- This equation shows that the resulting function is a straight line with a $slope = (1 - n)$.

- This shows clearly that it is possible, even easier, to achieve efficient parallel performance than is implied by Amdahl's speedup formula. Speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size.

1.3 Scalability of Parallel Architectures

- A parallel architecture is said to be scalable if it can be expanded (reduced) to a larger (smaller) system with a linear increase (decrease) in its performance (cost). This general definition indicates the desirability for providing equal chance for scaling up a system for improved performance and for scaling down a system for greater cost-effectiveness and/or affordability.

- Scalability is used as a measure of the system's ability to provide increased performance, for example, speed as its size is increased. In other words, scalability is a reflection of the system's ability to efficiently utilize the increased processing resources.

- The scalability of a system can be manifested as in the forms; *speed*, *efficiency*, *size*, *applications*, *generation*, and *heterogeneity*.

- **In terms of speed**, a scalable system is capable of increasing its speed in proportion to the increase in the number of processors.
  - Consider, for example, the case of adding m numbers on a 4-cube ($n = 16$ processors) parallel system.
  - Assume for simplicity that m is a multiple of n. Assume also that originally each processor has $(m = n)$ numbers stored in its local memory. The addition can then proceed as follows.
    * First, each processor can add its own numbers sequentially in $(m = n)$ steps. The addition operation is performed simultaneously in all processors.
    * Secondly, each pair of neighboring processors can communicate their results to one of them whereby the communicated result is added to the local result.
    * The second step can be repeated $(log_2 n)$ times, until the final result of the addition process is stored in one of the processors.
    * Assuming that each computation and the communication takes *one unit time* then the time needed to perform the addition of these m numbers is

      $$T_p = (m/n) + 2 * log_2 n$$

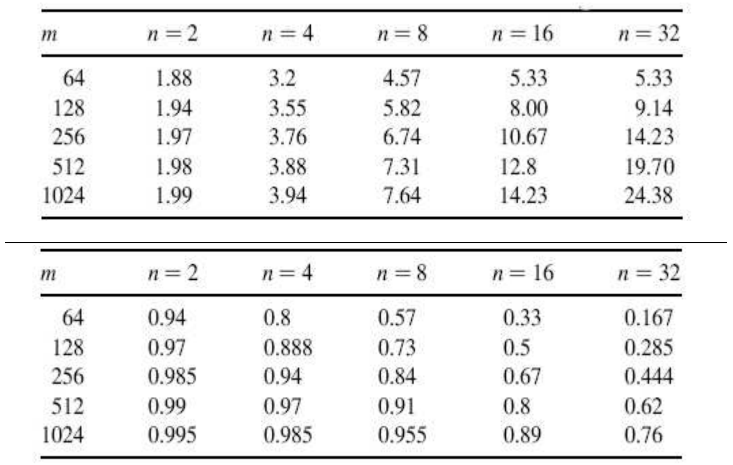
    * Recall that the time required to perform the same operation on a single processor is $T_s = m$. Therefore, the speedup is given by

      $$S = \frac{m}{(m/n) + 2 * log_2 n}$$

  - Figure 3a provides the speedup S for different values of m and n. It is interesting to notice from the table that for the same number of processors, n, a larger instance of the same problem, m, results in an increase in the speedup, S. This is a property of a scalable parallel system.

- **In terms of efficiency**, a parallel system is said to be scalable if its efficiency can be kept fixed as the number of processors is increased, provided that the problem size is also increased.
  - Consider, for example, the above problem of adding m numbers on an n-cube. The efficiency of such a system is defined as the ratio between the actual speedup, S, and the ideal speedup, n.

    $$E = \frac{S}{n} = \frac{m}{m + 2n * log_2 n}$$



m	$n=2$	$n=4$	$n=8$	$n=16$	$n=32$
64	1.88	3.2	4.57	5.33	5.33
128	1.94	3.55	5.82	8.00	9.14
256	1.97	3.76	6.74	10.67	14.23
512	1.98	3.88	7.31	12.8	19.70
1024	1.99	3.94	7.64	14.23	24.38

m	$n=2$	$n=4$	$n=8$	$n=16$	$n=32$
64	0.94	0.8	0.57	0.33	0.167
128	0.97	0.888	0.73	0.5	0.285
256	0.985	0.94	0.84	0.67	0.444
512	0.99	0.97	0.91	0.8	0.62
1024	0.995	0.985	0.955	0.89	0.76

Figure 3: The Possible Speedup and Efficiency for Different m and n.

- Figure 3b shows the values of the efficiency, E, for different values of m and n. The values in the table indicate that for the same number of processors, n, higher efficiency is achieved as the size of the problem, m, is increased.
- However, as the number of processors, n, increases, the efficiency continues to decrease.
- Given these two observations, it should be possible to keep the efficiency fixed by increasing simultaneously both the size of the problem, m, and the number of processors, n. This is a property of a scalable parallel system.
- It should be noted that the degree of scalability of a parallel system is determined by the rate at which the problem size must increase with respect to n in order to maintain a fixed efficiency as the number of processors increases.
    - For example, in a highly scalable parallel system the size of the problem needs to grow linearly with respect to n to maintain a fixed efficiency.
    - However, in a poorly scalable system, the size of the problem needs to grow exponentially with respect to n to maintain a fixed efficiency.
- It is useful to indicate at the outset that typically an increase in the speedup of a parallel system (benefit), due to an increase in

the number of processors, comes at the expense of a decrease in the efficiency (cost).

- In order to study the actual behavior of speedup and efficiency, we need first to introduce a new parameter, called the average parallelism Q.
  - It is defined as the average number of processors that are busy during the execution of given parallel software (program), provided that an unbounded number of processors are available.
  - The average parallelism can equivalently be defined as the speedup achieved assuming the availability of an unbounded number of processors.
- It has been shown that once Q is determined, then the following bounds are attainable for the speedup and the efficiency on an n-processor system:

$$\begin{array}{l} S(n) \geq \frac{nQ}{n+Q-1} \quad lim_{Q\to\infty} S(n) = n \quad lim_{n\to\infty} S(n) = Q \\ E(n) \geq \frac{Q}{n+Q-1} \end{array} \tag{9}$$

- The above two bounds show that the sum of the attained fraction of the maximum possible speedup, $S(n)/Q$, and attained efficiency, must always exceed 1.
- Notice also that, given a certain average parallelism, Q, the efficiency (cost) incurred to achieve a given speedup is given by

$$E(n) \geq \frac{Q - S(n)}{Q - 1}$$

- It is therefore fair to say that the average parallelism of a parallel system, Q, determines the associated speedup versus efficiency tradeoff.

- **Size scalability**; measures the maximum number of processors a system can accommodate. For example, the size scalability of the IBM SP2 is 512, while that of the symmetric multiprocessor (SMP) is 64.

- **Application scalability**; refers to the ability of running application software with improved performance on a scaled-up version of the system.
  - Consider, for example, an n-processor system used as a database server, which can handle 10,000 transactions per second. This

system is said to possess application scalability if the number of transactions can be increased to 20,000 using double the number of processors.

- **Generation scalability**; refers to the ability of a system to scale up by using nextgeneration (fast) components.
  - The most obvious example for generation scalability is the IBM PCs. A user can upgrade his/her system (hardware or software) while being able to run their code generated on their existing system without change on the upgraded one.
- **Heterogeneous scalability**; refers to the ability of a system to scale up by using hardware and software components supplied by different vendors.
  - For example, under the IBM Parallel Operating Environment (POE) a parallel program can run without change on any network of RS6000 nodes; each can be a low-end PowerPC or a high-end SP2 node.
- In his vision on the scalability of parallel systems, Gordon Bell has indicated that in order for a parallel system to survive, it has to satisfy five requirements. These are size *scalability*, *generation scalability*, *space scalability*, *compatibility*, and *competitiveness*. As can be seen, three of these long-term survivability requirements have to do with different forms of scalability.
- As can be seen, *scalability*, regardless of its form, is a desirable feature of any parallel system. This is because it guarantees that with sufficient parallelism in a program, the performance, for example, speedup, can be improved by including additional hardware resources without requiring program change.
- Owing to its importance, there has been an evolving design trend, called design for scalability (DFS), which promotes the use of scalability as a major design objective. Two different approaches have evolved as DFS. These are *overdesign* and *backward compatibility*.
  - Using the first approach, systems are designed with additional features in anticipation for future system scale-up. An illustrative example for such approach is the design of modern processors with 64-bit address,that is, 2^{64} bytes address space. Assume that

the current UNIX operating system supports only 32-bit address space. With memory space overdesign, future transition to 64-bit UNIX can be performed with minimum system changes.

– The other form of DFS is the backward compatibility. This approach considers the requirements for scaled-down systems. Backward compatibility allows scaled-up components (hardware or software) to be usable with both the original and the scaled-down systems. As an example, a new processor should be able to execute code generated by old processors. Similarly, a new version of an operating system should preserve all useful functionality of its predecessor such that application software that runs under the old version must be able to run on the new version.

1.4 Assignment:

Solve 2 questions.

1. Starting from the equation for the speedup factor given by

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

show the inequality that relates the fraction of serial computation, f, and the number of processors employed, n, if a 50% efficiency is to be achieved.

2. Consider a parallel architecture built using processors each capable of sustaining 0.5 megaflop. Consider a supercomputer capable of sustaining 100 megaflops. What is the condition (in terms of f) under which the parallel architecture can exceed the performance of the supercomputer?

3. Consider an algorithm in which $(1/\alpha)$ th of the time is spent executing computations that must be done in a serial fashion. What is the maximum speedup achievable by a parallel form of the algorithm?