transfers but only with all transfers in the same direction at any instant. For example, suppose a north direction is required. Each processing element can send information on the north-east diagonal path and accept information on the south-east diagonal path. For SIMD operations, all transfers in one step are in the same direction.

We will not investigate these systems further, and will leave array and vector computers to concentrate on MIMD computer systems. The reader is directed to the references quoted for further information on array computers.

6.4 General purpose (MIMD) multiprocessor systems

6.4.1 Architectures

In a general purpose MIMD computer system, a number of independent processors operate upon separate data concurrently. Hence each processor has its own program memory or has access to program memory. Similarly, each processor has its own data memory or access to data memory. Clearly there needs to be a mechanism to load the program and data memories and a mechanism for passing information between processors as they work on some problem. There are several possible architectures for general purpose MIMD multiprocessor systems. First, we can divide the multiprocessor systems into two types:

1. Shared memory multiprocessor systems.
2. Message-passing multiprocessor systems (without shared memory).

Shared memory multiprocessors use the centralized memory for holding data and communication purposes. Given that each processor must connect to program memory and data memory, most (fully) shared memory architectures can be represented by a set of processors, perhaps with local memory, connecting to one or more shared memory modules as shown in Figure 6.5. This is occasionally referred to as a "dance-hall architecture" as the processors are on one side of the network and the memories on the other side. We shall see there are many variations of interconnection networks, not all dance-halls. However, shared memory means that all processors have access to a common memory space. Hence location 100 (say) could be accessed by any processor. In any shared memory multiprocessor system, cache memory may be incorporated into the system. Cache memory creates significant complications into the design. We shall consider caches in shared memory multiprocessors in Chapter 7, Section 7.8. Also virtual memory may be incorporated into the system so that each processor must first translate a virtual address into a real address before passing the request to the memories.

A *message-passing multiprocessor* uses direct links to pass data between processors/processing elements and only have local memory (memory with each processor not directly accessible by other processors). A message-passing multiprocessor can be represented by Figure 6.6. Message-passing multiprocessors are sometimes called *multicom-*

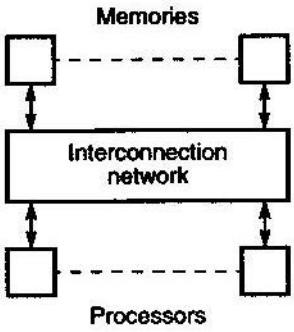


Figure 6.5 Shared memory multiprocessor model

puters because each processor/memory pair can be a separate computer. There are various possible direct link (*static*) interconnection networks for message-passing multiprocessor systems. A very restricted static interconnection network, but a particularly suitable scheme for VLSI fabrication, is to connect processors directly to their nearest neighbors, perhaps to other processors, in a two-dimensional array of processors. There are many other possible static interconnection networks. A major aspect of a message-passing multiprocessor system is how to map programs or processes onto individual processors and how to pass data that must be accessed by individual processors. Generally shared memory multiprocessors are easier to program and more flexible, but do not scale as well (not as easy to enlarge). Part III concentrates upon message-passing multiprocessor systems. In Part II, we concentrate upon shared memory multiprocessor systems.

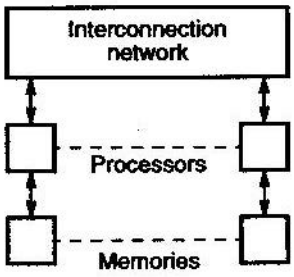


Figure 6.6 Message-passing multiprocessor model

6.4.2 Shared memory multiprocessor systems

In a shared memory multiprocessor system, different interconnection networks lead to systems with one of two characteristics:

1. Shared memory architecture with a uniform memory access (UMA).
2. Distributed shared memory architecture with non-uniform memory access (NUMA).

A third possibility exists, using only cache memory. This is known as a cache-only memory architecture (COMA).

Shared memory with a uniform memory access (UMA)

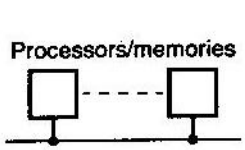
The uniform memory architecture is the direct implementation of the shared memory principle – a centralized shared memory is provided which is accessible by all processors via an interconnection network such that the path length and access time are about the same for all processors to all memory locations, just as a normal single processor system provides for equal access time to any main memory location. Uniform memory architectures can be described as consisting of a set of processors, perhaps with local cache memory connecting to one or more shared memory modules, as shown in Figure 6.5. If there is equal access to peripherals, the system is called a *symmetric multiprocessor*, otherwise it is an *asymmetric multiprocessor*.

Some UMA interconnection methods are shown in Figure 6.7. The single bus system (Figure 6.7(a)) is particularly convenient as a multiprocessor extension to a normal single processor microprocessor system, or other computer systems. In a single bus system with more than one processor attached to the bus, individual processors can access any of the memory modules attached to the bus, though only one data or instruction transfer can occur on the bus at any instant. The single bus system has been taken up by microprocessor manufacturers for multiple processor operation, and there are several standard buses which can support more than one microprocessor.

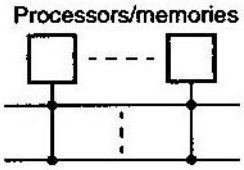
The multiple bus multiprocessor architecture shown in Figure 6.7(b) is a direct extension of the single bus architecture but with more than one bus connecting to all of the processors, memory modules and input/output interfaces. Processors can use any free bus to make a connection to a memory, but only B such connections can be made simultaneously, given B buses. Arbitration logic is necessary to resolve simultaneous requests, first to select up to one request for each memory module and then to select up to B of those requests to use the buses.

In the crossbar switch system (Figure 6.7(c)), a direct path is made between each processor and each memory module using one electronic bus switch to interconnect the processor and memory module. Each bus switch connects the set of processor bus signals, perhaps between forty and eighty signals, to a memory module. The crossbar architecture eliminates bus contention completely, though not memory contention, and can allow processors and memory to operate at their maximum speed.

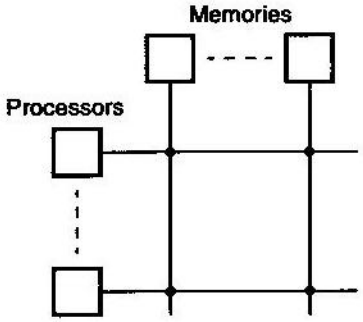
The multiport memory architecture, as shown in Figure 6.7(d), uses one multiport memory connecting to all the processors. Multiport memory is designed to enable more



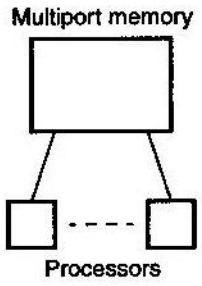
(a) Single bus



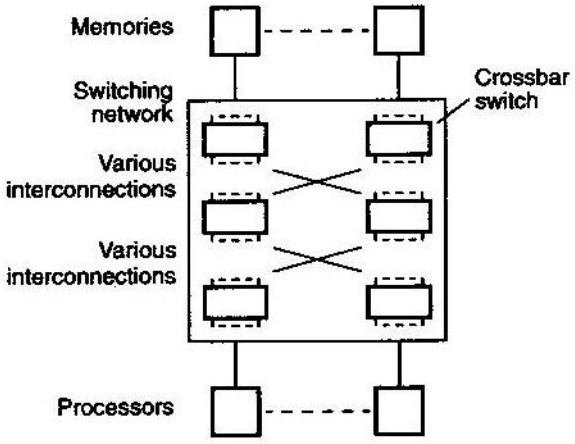
(b) Multiple buses



(c) Crossbar switch



(d) Multiport memory



(e) Multistage networks

Figure 6.7 Shared memory architectures

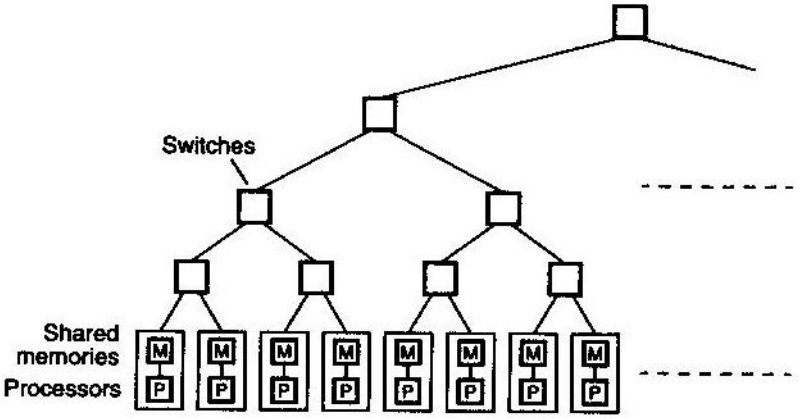
than one memory location to be accessed simultaneously, in this case by different processors. If there are, say, sixteen processors, sixteen ports would be provided into the memory, one port for each processor. Though large multiport memory could be designed, the design is too complex and expensive and consequently "pseudomultiport" memory is used, which appears to access more than one location simultaneously but in fact accesses the locations sequentially at high speed. Pseudomultiport memory can be implemented using normal single-port high speed random access memory with the addition of arbitration logic at the memory/processor interface to allow processors to use the memory on a first-come first-served basis. Using normal memory components, it is necessary for the memory to operate substantially faster than the processors. To service N simultaneous requests, the memory would need to operate at least N times faster than when servicing a single request. The multiport architecture with pseudomultiport memory can be considered as a variation of the crossbar switch architecture, with each column of crossbar switches moved to be close to the associated memory module.

The cost and complexity of the crossbar switch grows as $O(N^2)$ where there are N processors and memory modules. Hence, the crossbar interconnection network would be unsuitable for large numbers of processors and memory modules. In such cases, a multistage network (Figure 6.7(e)) can be used to reduce the number of switches. In such networks, a path is established through more than one switching element in an array of switching elements. Most multistage networks have three or more stages and each path requires one switch element at each stage.

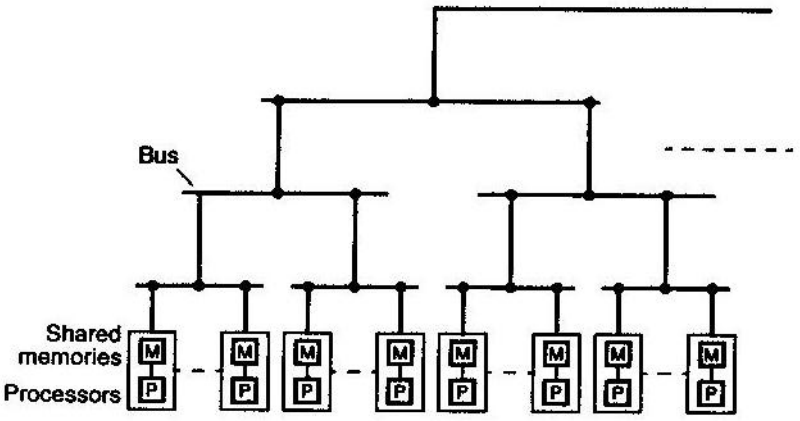
Distributed shared memory with non-uniform memory access (NUMA)

In the distributed shared memory system, shared memory modules are distributed among the processors so that each processor has attached a part of the total shared memory of the system. Again each processor could (and would normally have) cache memory. It is important to note that the memory has a single address space so that any processor could access any memory location directly by using its (real) address. Once we distribute the memory modules around the system, the access time to modules will depend upon the distance to the processor, i.e., we have a non-uniform memory access (time) architecture (NUMA).

A NUMA system will essentially be created by the use of a network in which the path length is longer for more distant destinations. Various networks are possible which have this characteristic, most of which are based upon a tree construction. Figure 6.8(a) shows the basic tree network as applied to a shared memory system. Here each processor has a memory which it can reach directly. All the memory in the system is reachable, but a processor must make a request through the network to make a connection to other memories. The network is made of switches which route the request to the desired memory. The route will be up until a downward path can be made, and then downwards. The path length will be a logarithmic function of the distance between the source and the destination. The primary motive for designing such a system is to aid scalability, as the tree can be extended relatively easily. Also some parallel algorithms have a tree structure



(a) Tree network



(b) Hierarchical bus network

Figure 6.8 NUMA networks

and these algorithms may map onto a tree network efficiently. Even if the algorithm does not have tree structure, many parallel algorithms have a locality in their communication/memory reference patterns – or can be designed with this is locality. Since the tree network favors locality of reference, algorithms with locality of reference will operate efficiently on the tree. A variation of the tree structure is the hierarchical bus structure shown in Figure 6.8(b). Here requests are first routed along the local bus and to higher level buses as needed to reach the destination. The hierarchical ring network is similar but with the buses replaced by paths formed as rings. The Kendall Square Research KRS1

had a hierarchical ring structure but with a COMA model (see next section). Other examples of NUMA networks include the reflective network (Chapter 8). Introducing separate cache memory with each processor into a system effectively creates non-uniform access. With non-uniform access, it is necessary to consider how best to distribute data and code for individual processors. It is almost a necessity that truly scalable systems employ a NUMA network.

Cache-only memory architecture (COMA)

Apart from the shared memory with a uniform memory access (UMA) and the shared memory with a non-uniform memory access (NUMA) architectures, a third possibility has been proposed, the cache-only memory architecture (COMA). In the COMA, each processor has a part of the shared memory, as in NUMA, but each part consists of cache memory, i.e., all the shared memory consists of cache memory. Therefore some of the cache memory could have data not being requested by the attached processor. COMA should cause data to migrate to the processor requesting it, just as in any multiprocessor having cache memory. It could be expensive to provide all memory as cache memory. Hagwersen, Landin and Haridi (1992) describe a COMA system having only some of the memory as true cache memory and the rest as normal memory which they call attraction memory. (Data is attracted to the attraction memory closest to the processor requesting it.) Other examples of COMA architectures include the Kendall Square Research KSR1 computer, and Swedish Institute of Computer Science's DDM system. Figure 6.9 shows the general COMA arrangement.

Fault tolerant systems

We mentioned in Section 6.1 that multiprocessor systems are sometimes designed to obtain increased reliability. The reliability of a system can be increased by adding redundant components. We can duplicate parts at the system level (extra systems), gate level (extra gates) or component level (extra transistors, etc.). To be able to detect failures and continue operating in the face of faults, the duplicate parts need to repeat actions

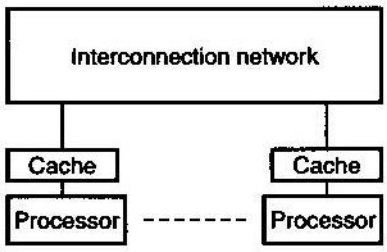


Figure 6.9 Cache-only memory architecture

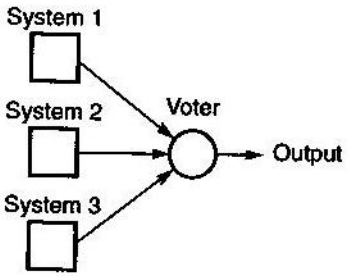


Figure 6.10 Triplicated system with a voter

performed by other parts, and some type of combining operating is performed which disregards the faulty actions. Alternatively, error detecting codes could be used; this requires extra gates.

One arrangement for system redundancy is to use three systems together with a voter circuit which examines the outputs of the systems, as shown in Figure 6.10. Each system performs the same computations. If all three systems are working, the corresponding outputs will be the same. If only two of the three systems are working, the voter chooses the two identical outputs. If more than one system is not working, the system fails. The probability that the system will operate is given by $P_s = P^3 + 3P^2(1 - P)$, i.e., the probability of all three systems operating or three combinations of two systems working and one not working. The triplicated system reliability is greater than for a single system during an initial operating period, but becomes less reliable later if the reliability decreases with time (see Problem 6.4). It is assumed that there is negligible probability of two faulty systems producing the same output, and that the voter will not fail. The concept can be extended to handle two faulty systems using five systems.

6.5 Potential for increased speed

To achieve an improvement in speed of operation through the use of parallelism, it is necessary to be able to divide the computation into tasks or processes which can be executed simultaneously. We might use a different computational algorithm with a multiprocessor rather than with a uniprocessor system, as it may not always be the best strategy simply to take an existing sequential computation and find the parts which can be executed simultaneously. Hence, a direct comparison is somewhat complicated by the algorithms chosen for each system. However, let us ignore this point for now. Suppose that a computation can be divided, at least partially, into concurrent tasks for execution on a multiprocessor system. A measure of relative performance between a multiprocessor system and a single processor system is the *speed-up factor*, $S(n)$, defined as:

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

which gives the increase in speed in using a multiprocessor. The (system) *efficiency*, E, is defined as:

$$E = \frac{S(n)}{n} \times 100\%$$

which gives how much processors are being used on the computation. If E = 50 per cent, the processors are being used half the time on the actual computation, on average. The maximum efficiency of 100 per cent occurs when all the processors are being used on the computation and the speed-up factor, $S(n)$, would be n.

There are various possible divisions of processes onto processors depending upon the computation, and different divisions lead to different speed-up factors. Also, any communication overhead between processors should be taken into account. Again, there are various possible communication overheads, from exhaustive communication between all processors to very limited communication between processors. The communication overhead is normally an increasing function of the number of processors. Here we will investigate some idealized situations. We shall use the term process to describe a contained computation performed by a processor; a processor may be scheduled to execute more than one process.

Equal duration process

The computation might be such that it can be divided into equal duration processes, with one process mapped onto one processor. This ideal situation would lead to the maximum speed-up of n, given n processors. The speed-up factor becomes:

$$S(n) = \frac{t}{t/n} = n$$

where t is the time on a single processor. Normally there would be communication between the processors. Suppose there is a communication overhead such that each process communicates once with one other process, but concurrently, as in a linear pipeline. The communications all occur simultaneously and thus appear as only one communication, as shown in Figure 6.11. Then the speed-up would be:

$$S(n) = \frac{t}{t/n + ct/n} = \frac{n}{1+c}$$

where c is the fractional increase in the process time which is taken up by communication between a pair of processes. If $c = 1$ then the time taken to communicate between

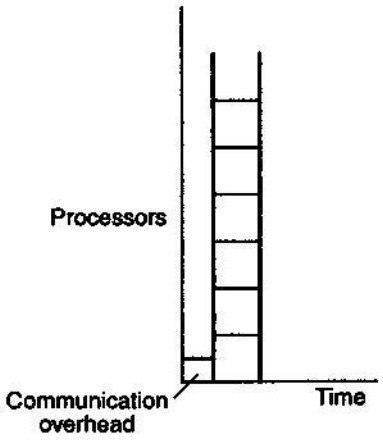


Figure 6.11 Equal duration tasks

processes is the same as the process time, $S(n) = n/2$, a reduction to half the speed-up.

In more general situations, the communication time will be a function of the number of processes and the communications cannot be fully overlapped.

Parallel computation with a serial section

It is reasonable to expect that some part of a computation cannot be divided at all into concurrent processes and must be performed serially. For example, the computation might be divided as shown in Figure 6.12. During some period, perhaps an initialization period or period before concurrent processes are being set up, only one processor is doing useful work and, for the rest of the computation, all of the available processors (n processors) are operating on the problem, i.e. the remaining part of the computation has been divided into n equal processes.

If the fraction of the computation that cannot be divided into concurrent tasks is f, and no overhead incurs when the computation is divided into concurrent parts, the time to perform the computation with n processors is given by $ft + (1 - f)t/n$ and the speed-up factor is given by:

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

This equation is known as *Amdahl's law*. Figure 6.13 shows $S(n)$ plotted against number of processors and plotted against f. We see that indeed a speed improvement is indicated, but the fraction of the computation that is executed by concurrent processes needs to be a substantial fraction of the overall computation if a significant increase in speed is to be

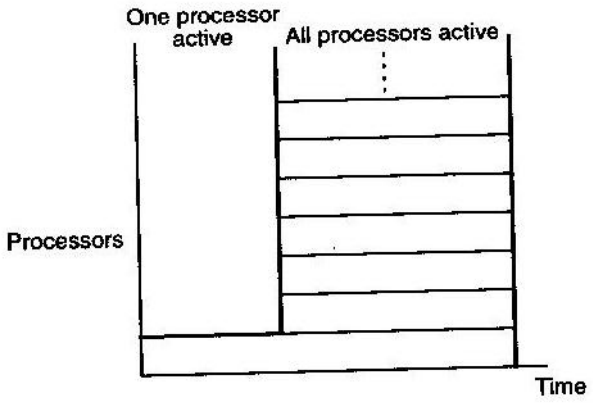


Figure 6.12 Parallel computation with serial section

achieved. The point made in Amdahl's law is that even with an infinite number of processors, the maximum speed-up is limited to $1/f$. For example, with only 5 per cent of the computation being serial, the maximum speed-up is 20, irrespective of the number of processors.

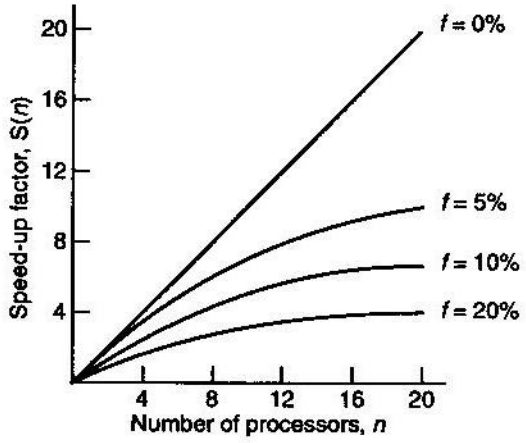
In fact, the situation could be worse. There will certainly be an additional computation to start the parallel section and general communication overhead between processes. In the general case, when the communication overhead is some function of n, say $tf_c(n)$, we have the speed-up given by:

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

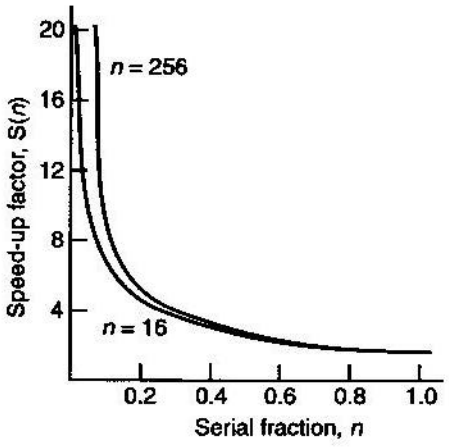
In practice we would expect computations to use a variable number of processors, as illustrated in Figure 6.14.

Optimum division of processes

We need to know whether utilizing all the available processors and dividing the work equally among processors is the best strategy, or whether an alternative strategy is better. Stone (1987) investigated this point and developed equations for different communication overheads, finding that the overhead eventually dominates, after which it is better not even to spread the processes among all the processors, but to let only one processor do the work, i.e., a single processor system becomes faster than a multiprocessor system. In our equations, this point is reached when the denominator of the speed-up equations equals or exceeds n, making $S(n)$ equal or less than one. Stone confirms that if dividing the process is best, spreading the processes equally among processors is best (assuming



(a) Speed-up against number of processors



(b) Speed-up against serial fraction, f

Figure 6.13 Speed-up factor

that the number of processes will divide exactly into the number of processors).

Speed-up estimates

It was once speculated that the speed-up is given by $\log_2 n$ (Minsky's conjecture). Lea (1986) used the term *applied parallelism* for the parallelism achieved on a particular system given the restricted parallelism processing capability of the system, and suggested that the applied parallelism is typically $\log_2 n$. He used the term *natural parallelism* for

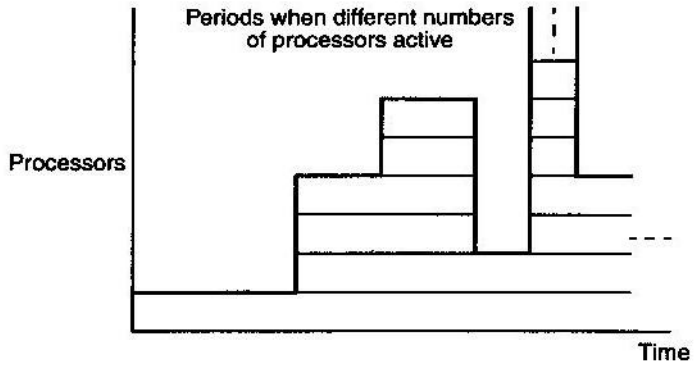


Figure 6.14 Parallel computation with variable processor usage

the potential in a program for simultaneous execution of independent processes and suggested that the natural parallelism is $n/\log_2 n$.

Hwang and Briggs (1984) presented the following derivation for speed-up being less than or equal to $n/\log_e n$. Suppose at some instant i processors are active and sharing the work equally with a load $1/i$ (seconds). Let the probability that i processors are active simultaneously be $P_i = 1/n$ where there are n processors. There is an equal chance of each number of processors ($i = 1, 2, 3 \ldots n$) being active. The (normalized) overall processing time on the multiprocessor is given by:

$$T_n = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{i}$$

The speed-up factor is given by:

$$S(n) = \frac{1}{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{i}} \leq \frac{n}{\log_e n} \leq \frac{n}{\log_2 n}$$

Figure 6.15 shows the speed-up estimates. If these values could not be improved upon in practice they would lead to the conclusion that multiprocessors give poor speed-up on large numbers of processors! The challenge is to disprove this statement in practical situations and with specific multiprocessor designs. In fact, some studies have achieved nearly perfect speed-up. For example, a 256-processor Butterfly multiprocessor system has achieved a speed-up of 230 on a range of numerical calculations (Rettberg and Thomas, 1986), whereas $n/\log_2 n$ gives a value of 32.

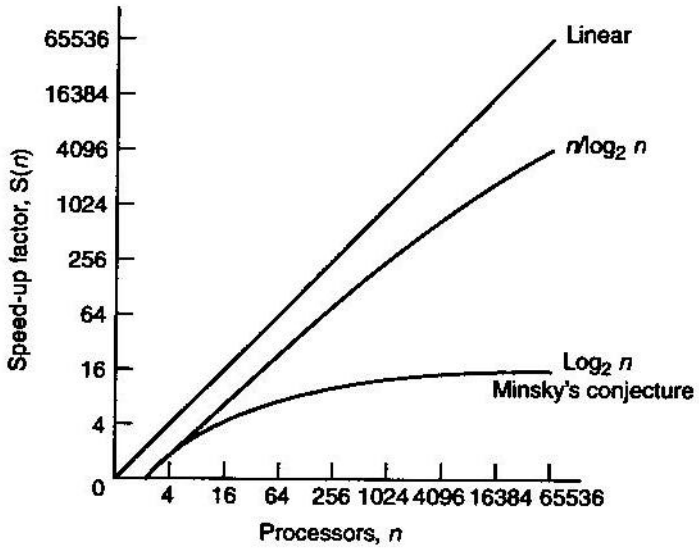


Figure 6.15 Speed-up factor estimates

Scalability

Scalability is a rather imprecise term used to indicate a design which allows the system to be increased in size and in doing so obtains increased performance. Of course we would want all multiprocessor systems to be scalable, but this will depend heavily upon the architecture of the system. Usually as we add processors to a system, the interconnection network must be expanded. Greater communication delays and contention results and the system efficiency, E, reduces. The underlying goal of most multiprocessor designs is to achieve scalability, and this is reflected in the multitude of interconnection networks which have been devised.

The scalability requirement can be relaxed by saying that increased performance only needs to be achieved with a larger problem size. For example, multiplication of 64×64 matrices might be performed on a 64 processor system in a time t, and multiplication of 256×256 matrices might be performed on a 256 processor system in time t in a (very) scalable system. This can be captured in a *scale up function*, $f(n)$, which is the actual execution time for n processors when n is increased proportionally to the size of the problem. This function indicates how well more processors can be employed with an increasing problem size. Ideally $f(n) = k$ where k is a constant. Scalability and the scale-up function are dependent upon the parallel algorithm being used. We need to use scalable parallel algorithms, algorithms in which the number of computational steps preferably increase faster than communication steps as the size of the problem increases.