1 Introduction to Parallel Processing

There is a number of concepts concerning parallel execution whose understanding is crucial in the following chapters, such as the notions of program, process, thread, concurrent and parallel execution, or kinds of concurrency and parallelism. Therefore, this chapter commences with the review of these notions. Subsequently, in the framework of a proposed classification, major parallel architecture classes will be introduced. This categorization can be seen as a major extension of Flynn's classic scheme, described also in the concluding part of this chapter.

1.1 Basic concepts

1.1.1 The concept of program

From the programmers perspective a program is an *ordered set of instructions*. On the other hand, from the *point of view of an operating system,* it is an *executable file* stored in the secondary (auxiliary) memory, typically on a disk. For example, Figure 1-1 shows the program P1.EXE as seen by the programmer and also by the operating system. The file, P1.EXE is stored in the secondary memory and can be accessed through the corresponding directory entry and file description.

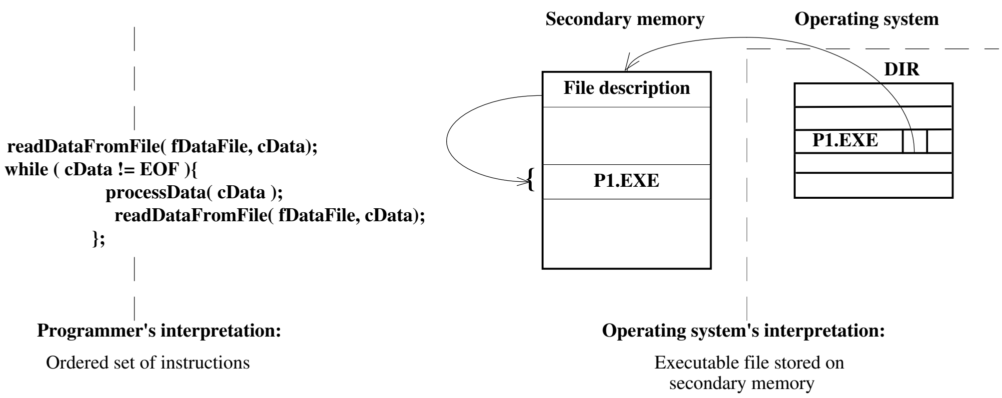


Figure 1-1. Dual interpretation of the notion program

1.1.2 The concept of process

In operating system terminology, the notion of **process** is used relating to execution instead of the term program. It designates a commission, or quantum of work dealt with as an entity. Consequently, required resources, like address space etc. will be allocated typically on a process basis.

Let's make a comparison. Imagine a car repair shop. Then a car that should be given in commission will be parked first on the parking place of the repair shop. This car resembles a program stored in the auxiliary memory. Subsequently, the owner goes to the counter where he or she asks the service station to repair the car. When the dispatcher accepts this request, a new commission will be created. Therefore, a work sheet will be filled in where all the relevant information pertaining to the commission will be recorded, such as registration number, owner's name etc. Afterwards, the repair commissions are represented by the work sheets and the dispatcher deals solely with the worksheets as far as scheduling of the work is concerned. Evidently, repair commissions correspond to the processes. Processes are created similarly and described by appropriate tables. Remaining by our example, in order to commence repairing a car, a depot and a mechanic should be allocated before. Furthermore, if additional resources, such as a hydraulic car lift, are required these also should be allocated to. Similarly, in case of process creation, memory space and, when needed, extra resources like I/O-devices should be allocated to a process. Lastly, to execute a process, a processor is to be allocated for.

This is expressed in operating system terminology by saying that the process is to be scheduled for execution.

Earlier operating systems, such as the IBM/360 operating systems like /DOS, /MFT, /MVT etc. used the term **task** in the same sense as recent operating systems use the notion of a process. Each process has a *lifecycle,* which consists of creation, an execution phase and termination.

In order to execute a program prior a corresponding process is to be created. A process will be brought into existence by using system services (system calls or supervisor macros). For example, in OS/2 processes will be created by means of the system call DosExecPgm, where the name of the executable program file will be given as a parameter like P1.EXE. The creation of a process means commissioning the operating system to execute a program.

Process creation involves the following four main actions:

- setting up the process description,
- allocation of an address space and
- loading the program into the allocated address space and
- passing on the process description to the scheduler.

Usually, Operating systems *describe* a process by means of a description table which we will call *Process Control Block* or *PCB*. Different operating systems are using, however, diverse designations for the process description table, like *Process Table* in UNIX, *Process Control Block* in VMS, *Process Information Block* in OS/2, or Task Control Block in IBM mainframe operating systems.

A PCB contains all the information that can be relevant during the whole lifecycle of a process. It holds on the one hand, *basic data*, such as process identification, owner, process status, description of the allocated address space etc. On the other hand it provides space for all implementation dependent, process specific *additional information.* Such supplementary information may be required sometimes during process management, in connection with memory management, scheduling etc., like page tables, working set lists, various timers relating to the execution of the process etc.. This may amount to a considerably extent.

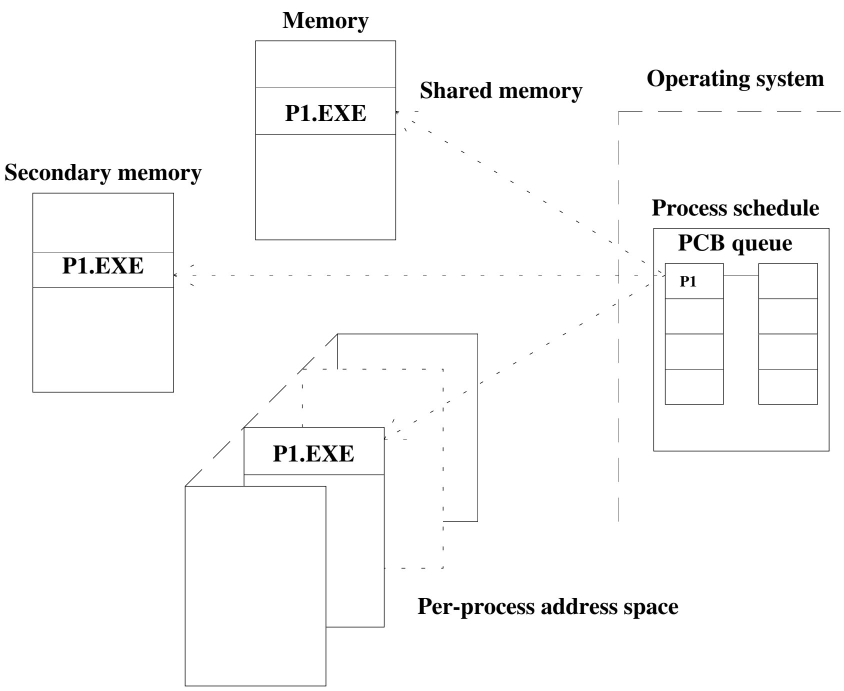


Figure 1-2. Description and location of a process after creation

A second major component of process creation is the **allocation of address space** to a process for execution. There are two basic ways of address space allocation, i.e. *sharing* the address space among the created processes or allocating distinct address spaces to each process (*per process address* *spaces*). Subsequently, the executable program file will usually be *loaded* into the allocated memory space. This will be accomplished corresponding to the memory management, either partly, when virtual memory management is used or, entirely, for real memory management.

Finally, a created process is to be *passed over to the process scheduler* that allocates the processor to the competing processes. The process scheduler manages processes typically by setting up and manipulating queues of PCBs. Thus, after creating a process the scheduler puts the PCB into a queue of ready-to-run processes.

Summing up, process creation results essentially in setting up the PCB, allocating a shared or a per process address space to the process, loading the program file and putting the PCB into the ready to run queue for scheduling (see Figure 1-2).

Nevertheless, there are *two submodels* for process creation to be mentioned. In the simpler submodel a process can not generate a new process, whereas in the more advanced one, a created process is allowed to establish new processes. Usually, this latter possibility is called **process spawning** (e.g. in UNIX or VMS) or **subtasking** (in IBM terminology).

Earlier operating systems, such as IBM/PCP or earlier versions of IBM/360/DOS, OS/MFT did not allowed process spawning. Thus, in such operating systems a program was represented as one process. In the more advanced submodel it is allowed for a process to create a new process, called *child process* (for instance in VMS or UNIX) or *subtask* (by IBM operating systems). Child processes can be created by the programmer using the standard mechanisms for process creation. Spawned processes form a *process hierarchy* termed also as **process tree** Figure 1-3 illustrates a program where process A spawned two processes B and C, and B further spawned another two processes, D and E.

Processes spawning results in *more concurrency* during execution. That is, in this case more processes belonging to the same program are competing for the processor and if the running process becomes blocked, another ready-to-run process pertaining to the same program can be scheduled to execution. However, the price of ensuring concurrency is the effort to create multiple processes and to carefully plan their communication and synchronization.

The execution phase of a process is under the control of the scheduler. It commences with creation of a process and lasts until the process is terminated. However, as far as scheduling is concerned, there are two basic models used by operating systems, termed as the *process model* and the *process-thread mode*. They differ essentially in the *granularity* of the units of work which are scheduled as one entity.

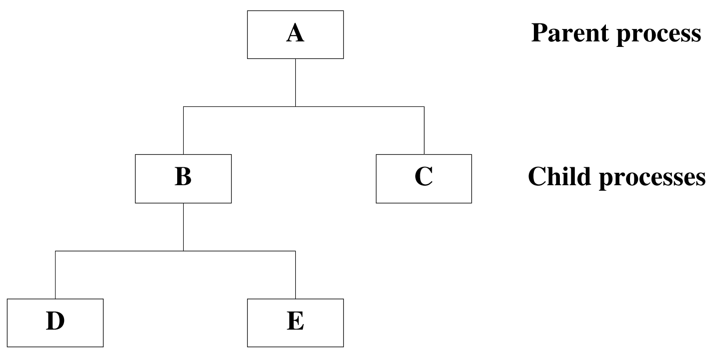


Figure 1-3. Process spawning

In the **process model** the scheduling is performed on a *per process* basis, i.e. the smallest unit of work to be scheduled is a process. On the contrary, the **process-thread model** is a *finer grained* scheduling model, where smaller units of work, called *threads*, are scheduled as entities. In the following we will outline the process model, whereas the process-thread model will be briefly discussed in connection with the description of threads.

Process scheduling involves three key concepts, the declaration of distinct **process states**, the specification of the **state transition diagram** and the statement of a **scheduling policy**.

As far as process states are concerned, there are *three basic states* connected with scheduling; the *ready to run* state, the *running* state as well as the *wait* (or *blocked*) state. In the **ready to run state** processes are able to run when a processor is allocated for them. In this state they are waiting for the processor to be executed. In the **running state** they are in execution on the allocated processor. In the

wait state they are suspended or blocked waiting for the occurrence of some events for getting again ready-to-run.

Based on the declared states the possible *state transitions* as well as their conditions are stated usually in a **state transition diagram**. For the quite straightforward case of the basic states, the state transition diagram of the scheduler could be specified as follows (see Figure 1-4).

When the scheduler selects a process for execution, its state will be changed from the ready-to-run to the running state. The process remains in this state until one of the following three events occur. Either it can happen that in compliance with the scheduling policy the scheduler decides to cease the execution of the particular process (for instance because the allocated time slice is over), and puts this process into the ready to run queue again, changing at the same time its state accordingly. Or the process in execution issues such an instruction that causes this process to wait till an event takes place. In this case the process state will be changed to the waiting state and its PCB will be placed into the queue of the waiting (or blocked) processes. Lastly, if the process arrives at the end of the execution, it terminates. Finally, a particular process, which is in the waiting state, can go over into the ready to run state, if the event it is waiting for has occurred. Real operating systems have a number of *additional states*, in the order of ten, introduced to cover features like swapping, different execution modes and so on.

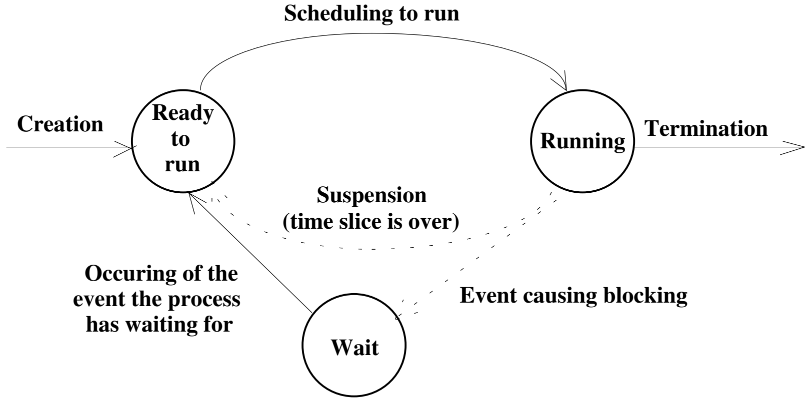


Figure 1-4. State transition diagram for the basic states

The last component of the scheduling specifies rules for the managing multiple competing processes which is usually termed as **scheduling policy**. Basic scheduling policies will be overviewed in connection with concurrent execution later in this chapter. By finishing the execution the process will be terminated releasing all the allocated resources. A large number of operating systems, in the first line earlier systems, is based on the process model (such as IBM/360 /DOS, OS/MFT, OS/MVT etc., UNIX, VMS).

1.1.3 The concept of thread

The notion of thread was introduced in the framework of the process-thread model in order to express more parallelism in code than in the process model. This will be achieved by declaration of smaller chunks of code, called *threads* (lightweight processes), within a process as an entity that can be executed concurrently or parallel. A **thread**, like a process, is a sequence of instructions. Threads are created *within and belonging to* processes. All the threads created within one process share the resources of the process, above all, the address space. Of course, scheduling will be performed on a per-thread basis. In other words, the process-thread model is a *finer grain scheduling model* than the process model.

Although this model is far more affordable then the process model, it has numerous *advantages* over and above. Evidently, with finer grained entities more parallelism can be exposed than in the case of the processes. In addition, the creation of threads or the communication, synchronization or switch among threads are far less expensive operations then those for processes, since all threads belonging to the same process are sharing the same resources. Therefore, most operating systems introduced in the 80s or 90s are based on the process-thread model, such as the OS/2, Windows NT or SunOS 5.0. Even recently, a standard is prepared to standardize the thread interface (IEEE POSIX

1003.40). Some operating systems, such as SunOS 5.0 are taking already into consideration this emerging new standard.

Threads have a similar *lifecycle* as the processes and will be managed mainly in the same way as processes are. Initially each process will be created with one single thread. However, threads are usually allowed to create new ones using particular system calls. For instance, in OS/2 new threads can be created by issuing the DoscCreateThread system call. Then, typically for each process a thread tree will be created (Figure 1-5).

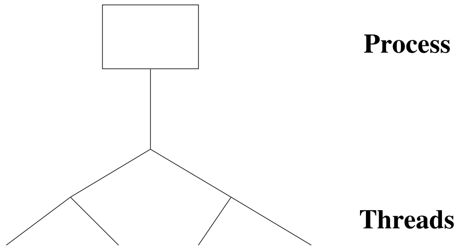


Figure 1-5. Thread tree

During creation each thread will be declared by a particular data structure mostly called *Thread Control Block* (*TCB*). The scheduling of threads will be performed in a similar way as described above for processes. Correspondingly, threads can be basically in one of *three states*; running, ready to run or waiting (blocked). Of course, each real operating system maintains, beyond the basic ones, a number of system specific additional states. For instance, Windows NT distinguishes the following additional states: initialized, terminated, standby (selected to execution but not yet in execution and transition state (a distinguished waiting state, marked by unavailability of resources). Again, a *state transition diagram* describes possible state transfers. Thread management is performed by setting up TCB queues for each state and performing the state transitions according to the state transition diagram and the scheduling policy. The scheduling part of the operating system overtakes the responsibility for managing all these queues in much the same way as it occurs with processes. At the end of thread creation the TCB will be placed into the queue of the ready-to-run threads and is contesting for the processor.

1.1.4 Processes and threads in languages

So far, processes and threads has been described as operating system entities that can be managed by system calls. However, there is a large number of programming languages, called concurrent and parallel languages, that enable to express parallelism at the language level by providing language tools to specify the creation of processes and threads. Moreover, these languages also contain language constructs to describe the synchronization and communication of processes and threads.

Like in operating systems, different languages use different terms (task, process, module, etc.) for processes and threads. Unfortunately, there is an ambiguity between processes of the operating systems and processes of the concurrent languages. Most of the concurrent and parallel languages (Ada, Concurrent Pascal, Occam-2, etc.) mean threads even if they use the term of tasks or processes. One of the rare exceptions is 3L Parallel C which use the name task to specify processes and the name thread to describe threads in compliance with the operating system terminology.

Concerning thread creation and termination there are three basic methods in concurrent languages:

- unsynchronized creation and termination,
- unsynchronized creation and synchronized termination,
- synchronized creation and termination.

The first method is typically realised by calling **library functions** as CREATE_PROCESS, START_PROCESS, CREATE_THREAD, and START_THREAD. As a result of these function calls, a new process or thread is created and started running independently from the parent one. The only connection between them is the possibility of communication and synchronization. However, when the child process or thread reaches its last instruction, it terminates without any synchronization to its parent.

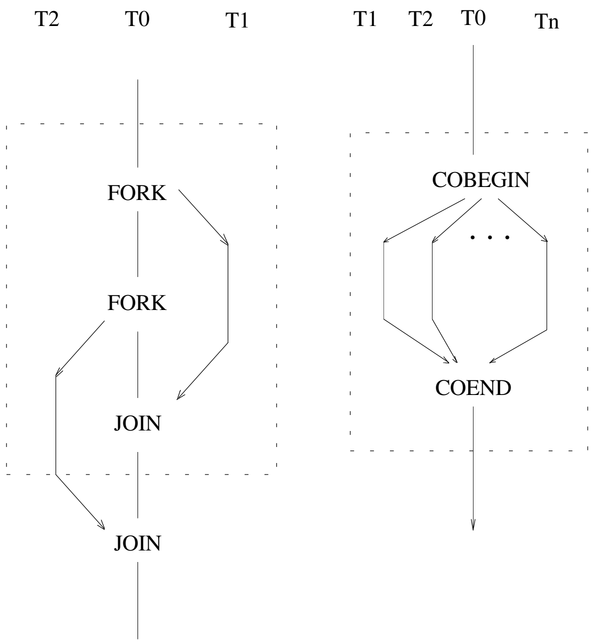
The second method relies on the use of two instructions: **FORK** and **JOIN**. The FORK instruction serves for spawning a new thread (or process) in order to deliver some computation result to the parent. When the parent needs the result of the child, it performs a JOIN instruction. At this point the two threads should be synchronized. The parent waits until the child reaches its terminating instruction and delivers the requested result.

The typical language construct to implement the third method is the **COBEGIN/COEND** structure:

.., COBEGIN, T1, T2, ..., Tn, COEND, ...

The threads between the COBEGIN/COEND brackets are executed in parallel. The work of the parent thread is suspended until all the child threads are terminated. At this point the parent can continue its progress by executing the instruction that follows COEND. A semantically equivalent construct called the **PAR** construct is used in the Occam-2 language.

The first two solutions represent an undisciplined programming style. Though they are more efficient than the third method, they can lead to unambigious process graphs, the verification of which is nearly impossible. However, compilers can use them in a disciplined way resulting in efficient parallel code. The situation is similar to the use of the GOTO instruction. Compilers can use it to realise higher level program constructs like loops, but their direct use by programmers led to unmaintainable software products.



(a) FORK/JOIN (b) COBEGIN/COEND

T: Thread

Figure 1-6. Thread graphs of language constructs to create and terminate threads

The comparison of the FORK/JOIN and COBEGIN/COEND constructs is illustrated by the thread graphs of Figure 1-6. The special characteristics of the COBEGIN/COEND construct is that it can always be included in a black box which has only one input and one output arc. This property enables reasonably easy debugging of individual threads in these programs. On the contrary, overlapped FORK/JOIN constructs prohibit this property rendering the verification of these programs difficult. The FORK/JOIN construct is a lower level one than the COBEGIN/COEND construct since the former can be used to implement the latter but not vice versa. Usually, lower level language constructs are more efficient to implement but more difficult and dangerous to use.

1.1.5 The concepts of concurrent and parallel execution

Although, the adjectives concurrent and parallel are often used as synonyms, it is often desirable to make a distinction between them.

Concurrent execution is the temporal behaviour of the *multiple clients* (requesters) *single server model* (Figure 1-7) where one single client is served at any given moment. This model has a dual nature, it is *sequential* in a diminutive time scale, but *simultaneous* in a rather large time scale.

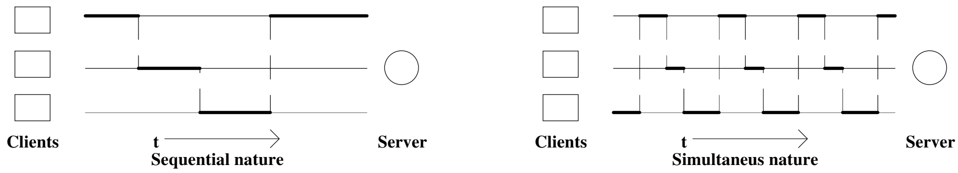


Figure 1-7. Single server client-server model with concurrent execution

In this situation the key problem is how the competing clients, let us say processes or threads, should be scheduled for service (execution) by the single server (processor). Scheduling policies may be oriented towards an efficient service in terms of highest throughput (least intervention) or towards short average respond time, etc..

The **scheduling policy** may be viewed as covering two aspects. The first one responds to the problem whether servicing of a client can be *interrupted* or not and, on what occasions (*preemption rule*). The other component states a rule how one of the competing clients will be *selected for* service (*selection rule*), see Figure 1-8.

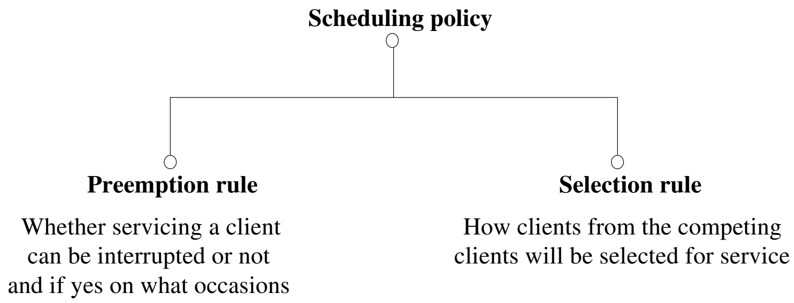


Figure 1-8. Main aspects of the scheduling policy

If pre-emption is not allowed, a client will be serviced as long as needed (Figure 1-9). It results often in intolerable long waiting times or in the blocking of important service requests for other clients. Therefore, a pre-emptive scheduling is often used. The **pre-emption rule** may specify either *time-sharing*, which restricts continuous service for each client merely for the duration of a time slice, or can be *priority based*, interrupting servicing one client whenever a higher priority client requests service, as shown in Figure 1-9.

The **selection rule** is typically based on some chosen parameters, such as priority, time of arrival etc.. This rule specifies an algorithm to determine a numeric value, which we will call *rank*, from the given parameters. During selection the ranks of all competing clients will be computed and the client with the *highest rank* will be scheduled for service. If more than one client have the same rank, an arbitration is needed to single out one (for example on a FIFO basis).

Parallel execution is associated with the *multiple clients multiple servers model* (Figure 1-10). Having more than one server (let us say processor) allows the servicing of more than one client (processes or threads) at the same time, which is called parallel execution.

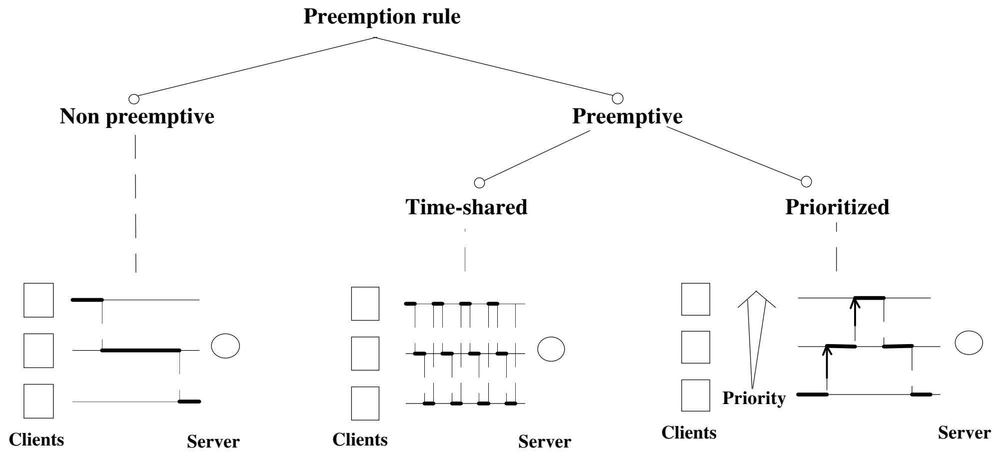


Figure 1-9. Basic preemption schemes

As far as the temporal harmonization of the executions is concerned there are two different schemes to be distinguished. In the **lock-step** or **synchronous scheme** each server starts service at the *same moment*, like in SIMD architectures. On the contrary, by the **asynchronous scheme**, the servers do not work concertedly, like in MIMD architectures.

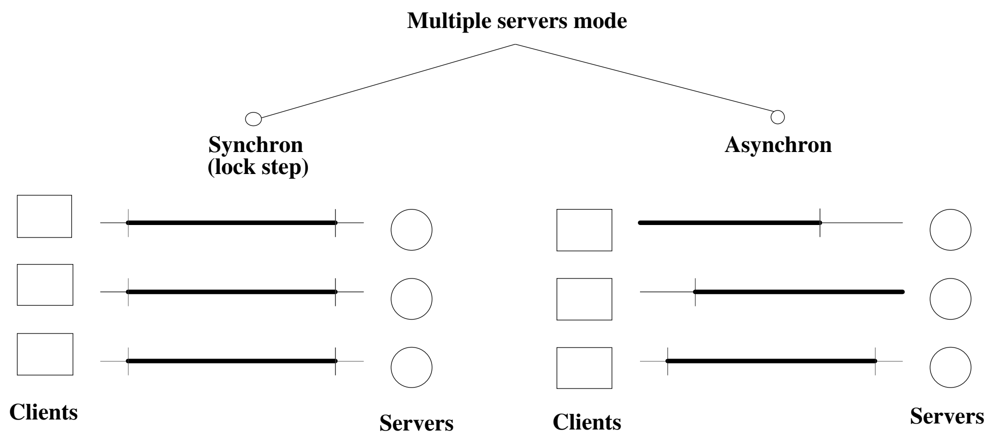


Figure 1-10. Multiple servers mode

1.1.6 Concurrent and parallel programming languages

Languages can be classified according to the available language constructs. Languages that do not contain any constructs to support the N-client model belong to the class of **sequential** (or traditional) **languages** (For example: C, Pascal, FORTRAN, Prolog, Lisp). **Concurrent languages** employ constructs to specify the N-client model by specifying parallel threads and processes but miss language constructs to describe the N-server model (For example: Ada, Concurrent Pascal, Modula-2, Concurrent Prolog). **Data parallel languages** introduce special data structures that are processed in parallel, element by element. They also apply special mapping directives to help the compiler in the optimal distribution of parallel data structures among processors. (For example: High Performance FORTRAN, DAP FORTRAN, DAP Prolog, Connection Machine C and Lisp). Finally, **parallel languages** extend the specification of the N-client model of concurrent languages with processor allocation language constructs that enable the use of the N-server model (For example: Occam-2, 3L

Parallel C, Strand-88). Table 1-1 summarizes the relationship between language classes and client-server models.

In Section 1.1.4 the constructs of concurrent languages supporting the N-client 1–server model have been described. Additionally to those constructs, parallel languages contain tools to specify the relation between processes and processors, i.e., the programmer can impose processor allocation on the compiler and run-time system.

A typical parallel language is Occam-2, which contains PAR constructs to create processes (clients) and channels to enable synchronization and communication among processes. These language features support the N-client model. Occam-2 also has a configuration part that does not affect the logical behaviour of the program but it enables processes to be arranged on the processors to ensure that performance requirements are met [INMOS88]. **PLACED PAR**, **PROCESSOR**, and **PLACE AT** commands are introduced for this purpose. In 3L Parallel C a separate **configuration language** is defined for similar purpose.

Languages	**1-client 1-server model**	**N-client 1-server model**	**1-client N-server model**	**N-client N-server model**
Sequential	+	-	-	-
Concurrent	+	+	-	-
Data parallel	+	-	+	-
Parallel	+	+	-	+

Table 1-1 Classification of programming languages

Though parallel languages contain more types of language constructs than concurrent languages, it does not mean that they are superior to concurrent or sequential languages. On the contrary, the configuration part of parallel languages represent an extra burden for the user. It is much more convenient for the programmer if the task of the configuration is performed either by the compiler or by the run-time system.

1.1.7 Concurrent execution models

Finally, in order to round up the picture we will below briefly outline high level concurrent execution models such as *multithreading, multitasking, multiprogramming* and *time sharing*, see also Figure 1-11). Note that all these concurrent execution models are referring to *different granularity* of execution termed also as different levels. Typically, they are implemented by means of the operating system on one single processor (that is on a SISD, in Flynn's taxonomy). It is important to note however, that all these notions can also be interpreted in a broader sense, as notions designating the execution of multiple threads, processes or users in a concurrent or parallel way. Nevertheless, the interpretations below are formulated focused on the narrower scope of concurrent execution.

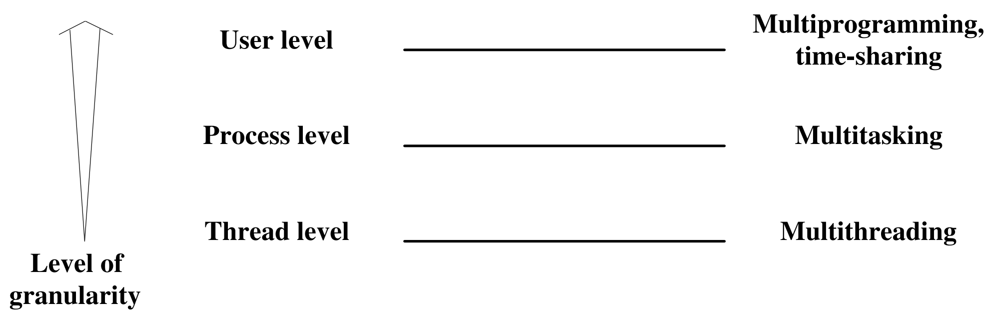


Figure 1-11. Concurrent execution models

Thread level concurrent execution is termed as **multithreading**. In this case multiple threads can be generated to each process, and these threads will be executed concurrently on a single processor under the control of the operating system.

Multithreading will usually be interpreted as *concurrent execution at the thread level*. Multithreading presumes evidently that multiple threads of a process exist, that is, a process-thread model is used to represent and schedule units of work for the processor. Multithreading is supported by recent operating systems (like OS/2, Windows NT or SunOS 5.0) as well as by multithreaded architectures.

Process level concurrent execution is usually called *multitasking*. Each widely used present operating system supports this concept. **Multitasking** refers to concurrent execution of processes. Multiple ready to run processes can be created either by a single user if process spawning is feasible, or by multiple users participating in multiprogramming or in time-sharing. Multitasking was introduced in operating systems in the middle 60s, among others, in the IBM operating systems for the System/36O such as /DOS, OS/MFT and OS/MVT. Almost all recently used operating systems provide this feature. MIMD architectures support this model of parallel execution.

Finally, user level concurrent or parallel execution can be either *multiprogramming* or *time-sharing*. **Multiprogramming** aims at the effective utilization of the processor by creating multiple ready-to-run processes, each belonging to different users. If the actually running process becomes blocked, because it has to wait for a particular event, such as completition of I/O, the processor will be scheduled to another ready to run process (if any). This mode of operation was supported by multiprogrammed operating systems such as the systems mentioned in the forgoing paragraph. Notice that multiprogramming is *internally* implemented as *multitasking* for *independent tasks* arising from different users.

On the other hand, **time-sharing** has the objective to offer *adequate quality* computer *services* to a number of users through terminals. This mode of operation intends to guarantee a *short access time* to all users instead of striving for the effective utilization of the processor. In this case a number of ready-to-run processes are created again, however, by different users. The scheduling has to be performed in such a way that a proper response time could be guaranteed for each user. Time-sharing evolved as a response to the lengthy turn around times of efficiency oriented and user unfriendly multiprogrammed systems in the 60's (TSS, MAC etc.).

1.2 Types and levels of parallelism

1.2.1 Available and utilised parallelism

Parallelism is one of the 'hottest' ideas in computing. Architectures, compilers and operating systems have been striving to extract and utilise as much parallelism as possible for more than two decades in order to speed-up computation,.

Related to our subject the notion of parallelism is used in two different contexts. Either it designates **available parallelism** in programs (or in a more general sense in problem solutions) or it

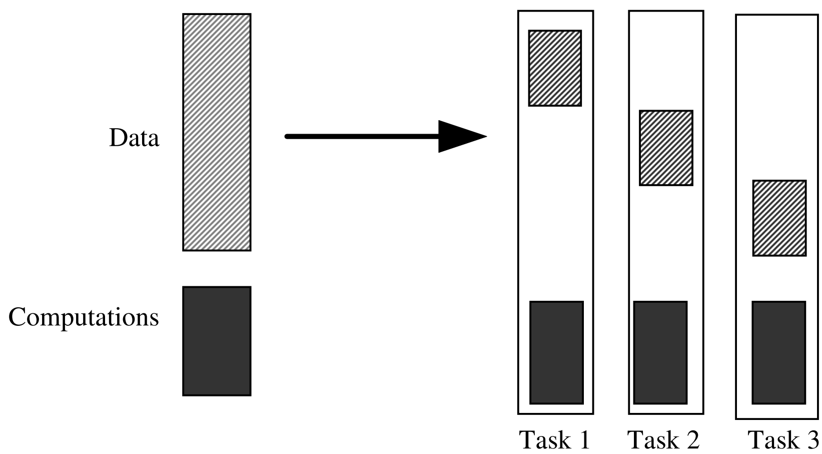


Figure 1-12 The principle of data parallelism

refers to parallelism occurring during execution, called **utilised parallelism**. First, let us overview available parallelism.

1.2.2 Types of available parallelism

Problem solutions may contain two different kinds of available parallelism, called data parallelism and functional parallelism.

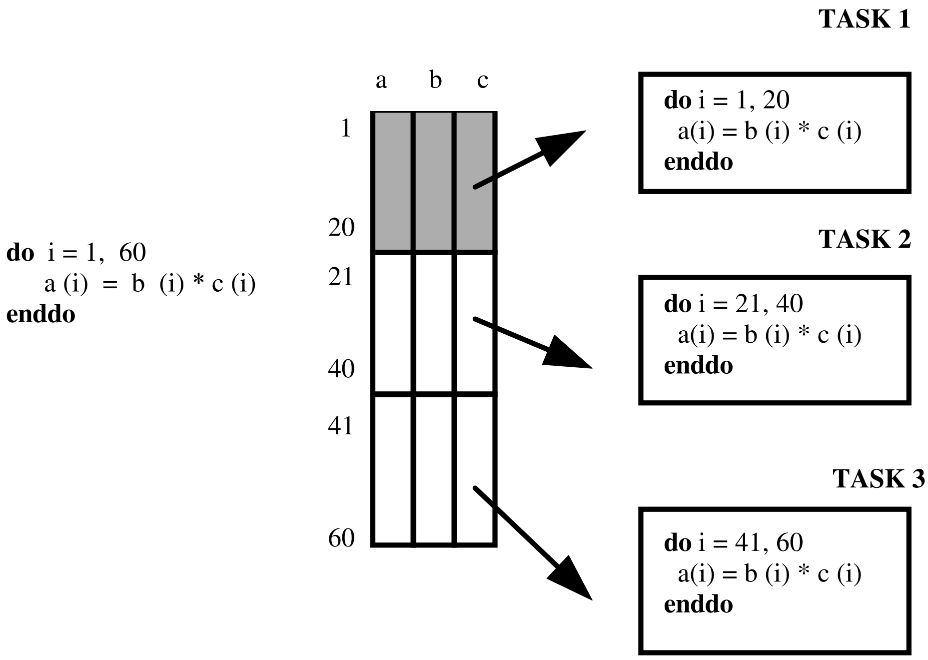


Figure 1-13 Data parallelism in do-loops

Data parallelism is the use of multiple functional units to apply the *same operation* simultaneously to *different elements of a data set*. A k-fold increase in the number of functional units leads to a k-fold increase in the throughput of the system, if there is no overhead associated with the increase in parallelism. In case of data parallelism, the level of achievable parallelism is proportional to the amount of data. Data parallelism is well suited for scientific computation on large and regular data structures (vectors, matrices, etc.). Data decomposition and the distribution of data segments among several processes (tasks) are shown in Figure 1-12.

A typical example to exploit data parallelism appears in do-loops as illustrated on a simple example in Figure 1-13. Another typical form of data parallelism is applied in solving Laplace's equations on a square based on the Jacobi iterative method. In such neighbourhood-oriented computations we get processes with a regular communication pattern. Figure 1-14 depicts a simple example for the Jacobi method using data parallel execution scheme.

Data parallelism originates in using such data structures in problem solutions, which allow parallel operations on their elements, such as vectors or matrices ([HillStee86]). Data parallelism is inherent only in a restricted set of problems, such as scientific or engineering calculations or image processing. Usually, this kind of parallelism gives rise to a massively parallel execution for the data-parallel part of the computation. Thus, the actual values for the achievable speed-up depend heavily on the characteristics of the application in concern.

We term **functional** that kind of **parallelism** which arises from the logic of a problem solution. It occurs more or less in all formal descriptions of problem solutions, like program flow diagrams, data flow graphs, programs etc. to a more or less extent. However, in the following we will restrict our attention to available functional parallelism inherent in programs expressed in traditional imperative languages.

In contrast to data parallelism, in which parallelism is achieved by decomposing data into pieces and applying the same operation to each element of the data set, **function parallelism** is achieved by applying *different operations (functions)* to different data elements simultaneously. Accordingly, it is the computation which is decomposed into pieces (functions) as shown in Figure 1-15.

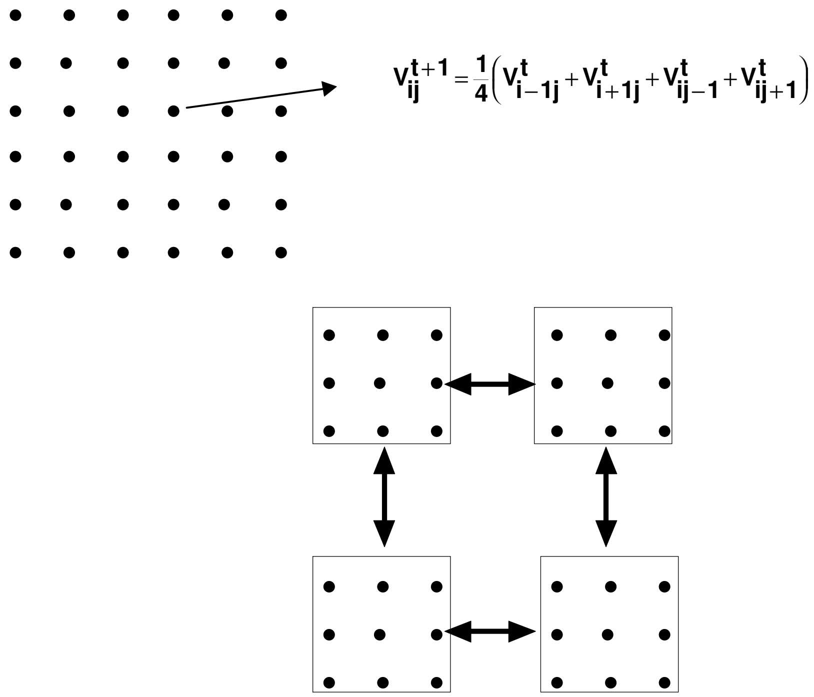


Figure 1-14 Jacobi iterative method for solving Laplace's equation on a square

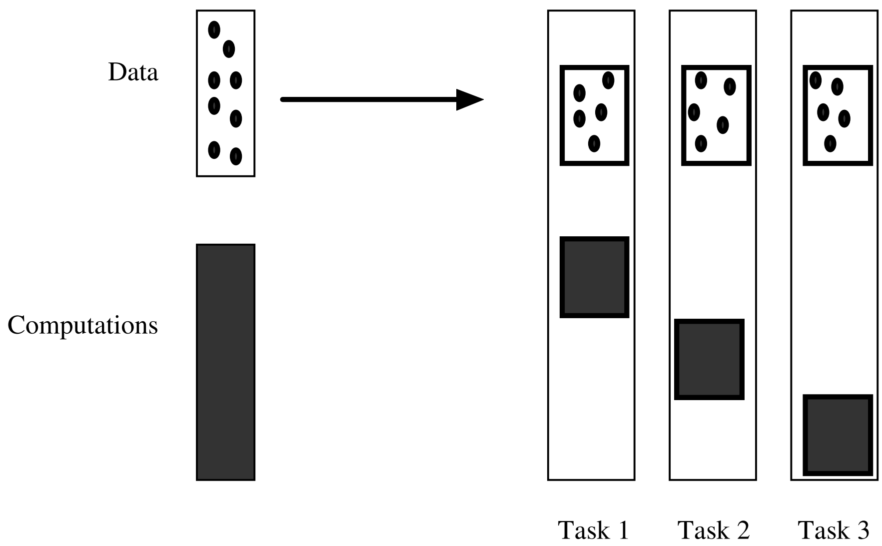


Figure 1-15 The principle of function parallelism

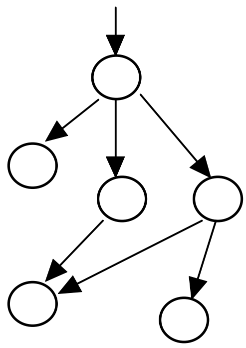


Figure 1-16 Data-flow graph to represent function parallelism

Data are taken to the processes where they are needed. The flow of data among these processes can be arbitrarily complex. The data-flow graph visualises the data dependencies among the different operations. Nodes of the graph represent operations (instructions or functions) while the directed arcs show the direction of data flow among the operations (see Figure 1-16).

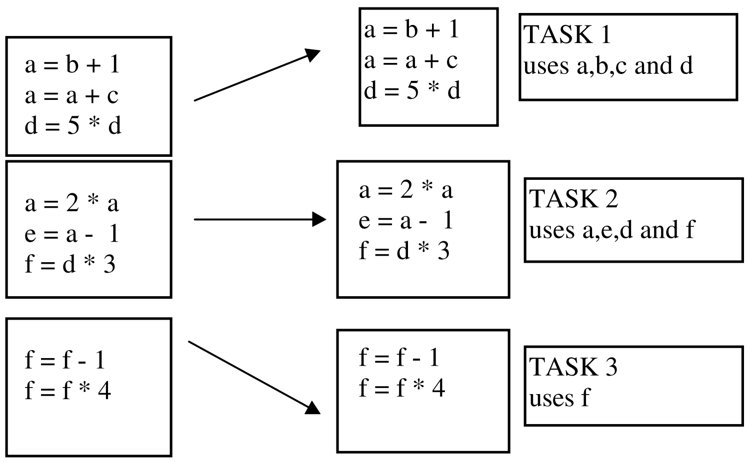


Figure 1-17 Function Parallelism: A simple example

A simple example for function parallelism is depicted in Figure 1-17. Function parallelism is suitable for non regular computations resulting in non regular communication pattern and the level of parallelism is usually limited by the number of functions.

If the data-flow graph forms a simple directed graph (like in the floating point adder of a vector computer as shown in Figure 1-19), then we say the algorithm is pipelined. A **pipelined computation** is divided into a number of steps called stages. Each stage works at full speed on a particular part of a computation. The output of one stage forms the input of the next stage (see Figure 1-18).

If all the stages work at the same speed, once the pipe is full the work rate of the pipeline is equal to the sum of the work rates of the stages. A pipeline is analogous to an assembly line: the flow of results is simple and fixed, precedence constraints must be honoured, and it takes time to fill and drain the pipe. If we assume that each stage of the pipe requires the same amount of time, the multiplicative increase in the throughput is equal to the number of stages in the pipeline which is usually very limited. Pipelining is suitable for regular computations on regular data. Figure 1-19 illustrates a typical pipelining example used in vector computers to add vectors.

From another point of view parallelism can be considered as being either regular or irregular. *Data parallelism* is *regular*, whereas *functional parallelism*, with the exception of loop-level parallelism, is usually *irregular*. Intuitively, parallelism will be called weak, when the extent of the available or exploited parallelism remains in the *one digit range*. It is the typical case for irregular parallelism. On the contrary, regular parallelism is often **massive**, offering several orders of magnitude in speed-up.

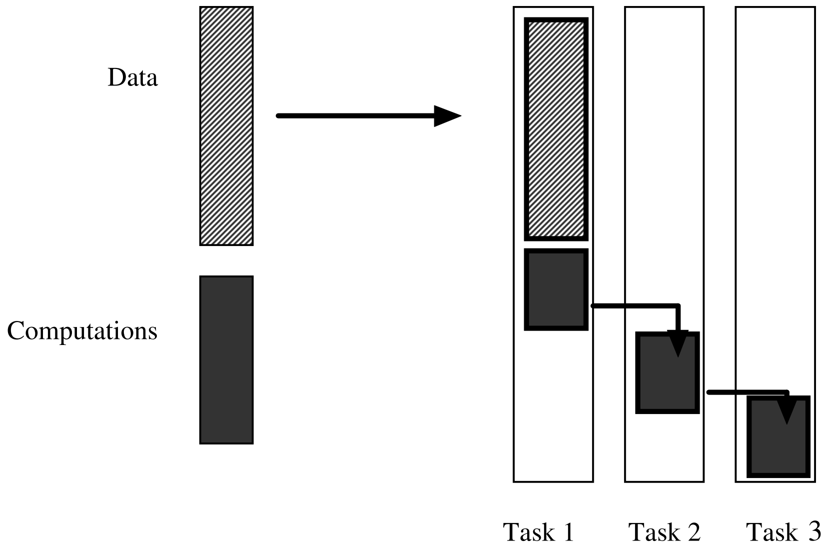


Figure 1-18 The principle of pipeline parallelism

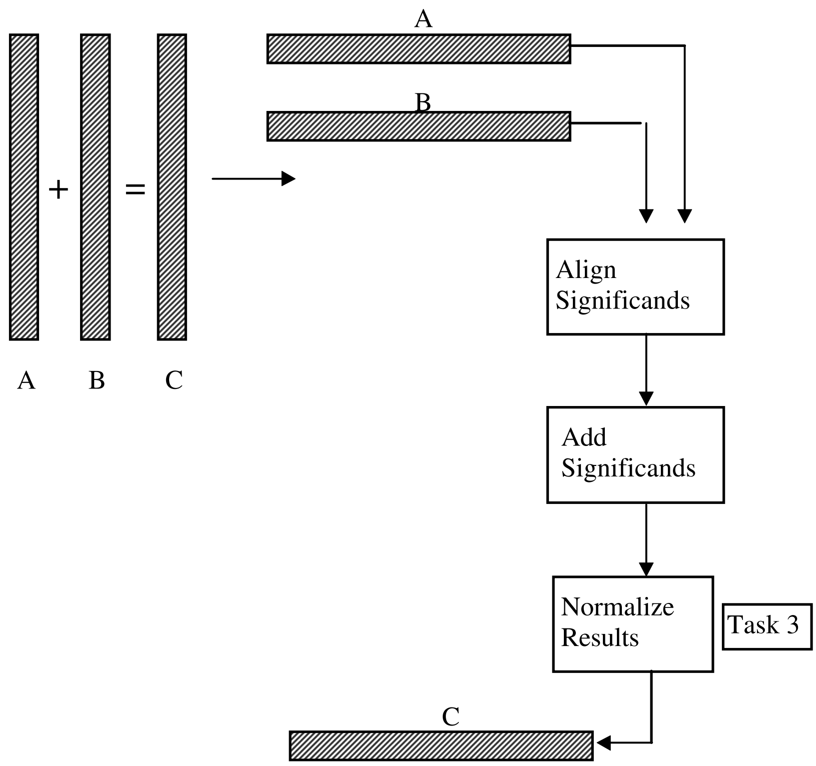


Figure 1-19 Structure of a pipeline adder in a vector processor

1.2.3 The Sieve of Eratosthenes

In this section we will explore methods to parallelize the Sieve of Eratosthenes, the classic prime-finding algorithm. We will design and analyse both function parallel, data parallel and pipeline implementations of this algorithm.

We want to find the number of primes less than or equal to some positive integer n. A prime number has exactly two factors: itself and 1. The Sieve of Eratosthenes begins with a list of natural numbers 2, 3, 4, ..., n, and removes composite numbers from the list by striking multiples of 2, 3, 5, and successive primes. The sieve terminates after multiples of the largest prime less than or equal to $\sqrt{n}$ have been struck.

A sequential implementation of the Sieve of Eratosthenes manages three key data structures: a boolean array whose elements correspond to the natural numbers being sieved, an integer corresponding to latest prime number found, and an integer used as a loop index incremented as multiples of the current prime are marked as composite numbers.

1.2.3.1 Function parallel approach

First let's examine a function parallel algorithm to find the number of primes less than or equal to some positive integer n. In this algorithm every processor repeatedly goes through the two-step process of finding the next prime number and striking from the list multiples of that prime, beginning with its square. The processors continue until a prime is found whose value is greater than $\sqrt{n}$. Using this approach, processors concurrently mark multiples of different primes. For example, one processor will be responsible for marking multiples of 2 beginning with 4. While this processor marks multiples of 2, another may be marking multiples of 3 beginning with 9.

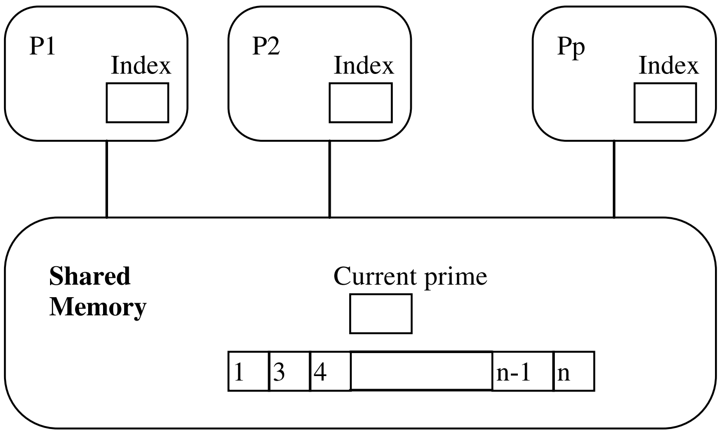


Figure 1-20 Shared-memory model for function parallel Sieve of Erathostenes algorithm

Each processor has its own private loop index and shares access to other variables with all the other processors

. We will base the function parallel algorithm on the simple model of parallel computation illustrated in Figure 1-20. Every processor shares access to the boolean array representing the natural numbers, as well as the integer containing the value of the latest prime number found. Because processors independently mark multiples of different primes, each processor has its own local loop index.

If a group of asynchronously executing processors share access to the same data structure in an unstructured way, inefficiencies or errors may occur. Here are two problems that can occur in the algorithm we just described. First, two processors may end up using the same prime value to sieve through the array. Normally a processor accesses the value of the current prime and begins searching at the next array location until it finds another unmarked cell, which corresponds to the next prime. Then it updates the value of the integer containing the current prime. If a second processor accesses the value of the current prime before the first processor updates it, then both processors will end up finding the same new prime and performing a sieve based on that value. This does not make the algorithm incorrect, but it wastes time.

Second, a processor may end up sieving multiples of a composite number. For example, assume processor A is responsible for marking multiples of 2, but before it can mark any cells, processor B finds the next prime to be 3, and processor C searches for the next unmarked cell. Because cell 4 has not yet been marked, processor C returns with the value 4 as the latest "prime" number. As in the previous example, the algorithm is still correct, but a processor sieving multiples of 4 is wasting time. In later chapters we will discuss ways to design parallel algorithms that avoid such problems.

For now, let's explore the maximum speedup achievable by this parallel algorithm, assuming that none of the time-wasting problems described earlier happen. To make our analysis easier, we will also ignore the time spent finding the next prime and concentrate on the operation of marking cells.

First let's consider the time taken by the sequential algorithm. Assume it takes 1 unit of time for a processor to mark a cell. Suppose there are k primes less than or equal to SQUARE(n). We denote these primes $\pi_1, \pi_2, \ldots, \pi_k$. (For example, $\pi_1 = 2$, $\pi_2 = 3$, and $\pi_3 = 5$.) The total amount of time a single processor spends striking out composite numbers is

$$\left\lceil \frac{(n+1)-\pi_1^2}{\pi_1} \right\rceil + \left\lceil \frac{(n+1)-\pi_2^2}{\pi_2} \right\rceil + \left\lceil \frac{(n+1)-\pi_3^2}{\pi_3} \right\rceil + \ldots + \left\lceil \frac{(n+1)-\pi_k^2}{\pi_k} \right\rceil$$
$$= \left\lceil \frac{n-3}{2} \right\rceil + \left\lceil \frac{n-8}{3} \right\rceil + \left\lceil \frac{n-24}{5} \right\rceil + \ldots + \left\lceil \frac{(n+1)-\pi_k^2}{\pi_k} \right\rceil$$

There are $\lceil (n-3)/2 \rceil$ multiples of 2 in the range 4 through n, $\lceil (n-8)/3 \rceil$ multiples of 3 in the range 9 through n, and so on. For n = 1,000 the sum is 1,411.

Now let's think about the time taken by the parallel algorithm. Whenever a processor is unoccupied, it grabs the next prime and marks its multiples. All processors continue in this fashion until the first prime greater than $\sqrt{n}$ is found.

For example, Figure 1-21 illustrates the time required by one, two, and three processors to find all primes less than or equal to 1,000. With two processors the parallel algorithm has speedup 2 (1,411/706). With three processors the parallel algorithm has speedup 2.83 (1,411/499). It is clear that the parallel execution time will not decrease if more than three processors are used, because with three or more processors the time needed for a single processor to sieve all multiples of 2 determines the parallel execution time. Hence an upper bound on the execution time of the parallel algorithm for n = 1,000 is 2.83.

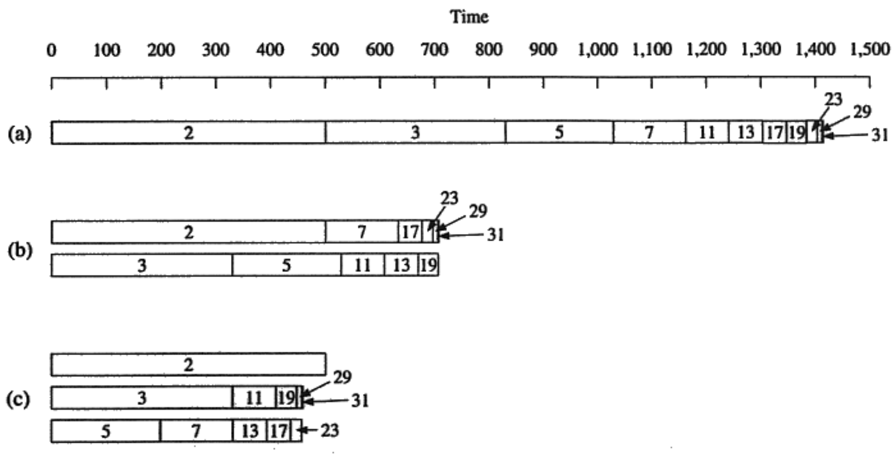


Figure 1-21 Study of how adding processors reduces the execution time of the function parallel Sieve of Eratosthenes algorithm when n = 1,000. The number in the bar represents the prime whose multiples are being marked. The length of the bar is the time needed to strike these multiples. (a) Single processor strikes out all composite numbers in 1,411 units of line. (b) With two processors execution time drops to 706 time units. (c) With three or more processors execution time is 499 time units, the time needed for a processor to strike all multiples of 2.

1.2.3.2 Data parallel approach

Let's consider another approach to parallelizing the Sieve of Eratosthenes. In our new algorithm, processors will work together to strike multiples of each newly found prime. Every processor will be responsible for a segment of the array representing the natural numbers. The algorithm is data parallel, because each processor applies the same operation (striking multiples of a particular prime) to its own portion of the data set.

Analysing the speedup achievable by the data-parallel algorithm on the shared memory model of Figure 1-20 is straightforward; we have left it as an exercise. Instead, we will consider a different model of parallel computation (Figure 1-22). In this model there is no shared memory, and processor interaction occurs through message passing.

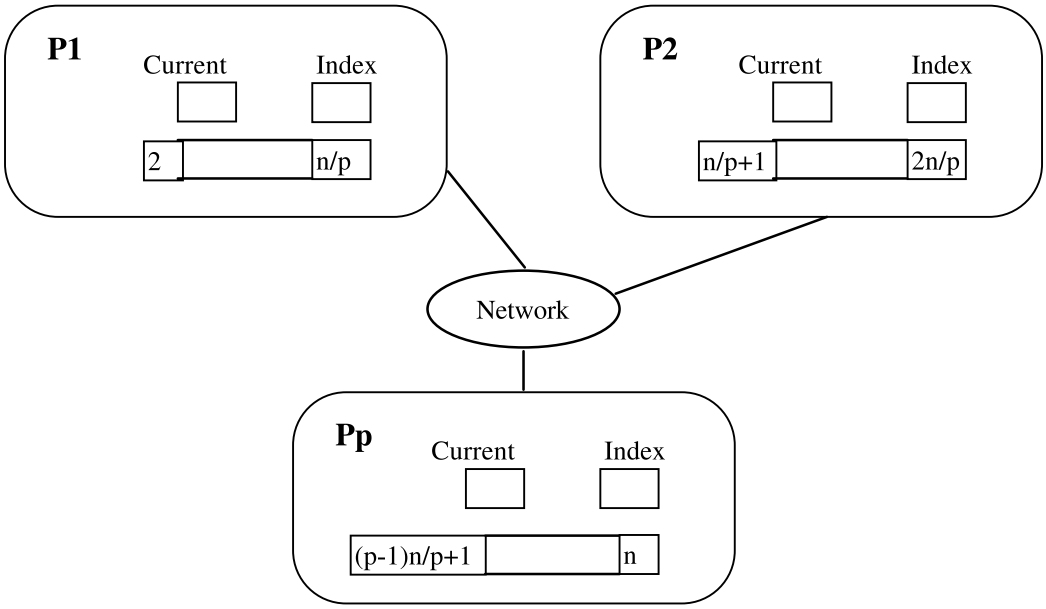


Figure 1-22 Private memory model for parallel Sieve of Eratosthenes algorithm. Each processor has its own copy of the variables containing the current prime and the loop index. Processor 1 finds primes and communicates them to the other processors. Each processor iterates through its own portion of the array of natural numbers, marking multiples of the prime

Assume we are solving the problem on p processors. Every processor is assigned no more than $\lceil n/p \rceil$ natural numbers. We will also assume that p is much less than SQUARE(n). In this case all primes less than SQUARE(n), as well as the first prime greater than SQUARE(n), are in the list of natural numbers controlled by the first processor. Processor 1 will find the next prime and broadcast its value to the other processors. Then all processors strike from their lists of composite numbers all multiples of the newly found prime. This process of prime finding and composite number striking continues until the first processor reaches a prime greater than SQUARE(n), at which point the algorithm terminates.

Let's estimate the execution time of this parallel algorithm. As in the previous analysis, we ignore time spent finding the next prime and focus on the time spent marking composite numbers. However, since this model does not have a shared memory, we must also consider the time spent communicating the value of the current prime from processor 1 to all other processors.

Assume it takes χ time units for a processor to mark a multiple of a prime as being a composite number. Suppose there are k primes less than or equal to SQUARE(n). We denote these primes $\pi 1$, $\pi 2$, . . ., πk. The total amount of time a processor spends striking out composite numbers is no greater than

$$\left(\left\lceil \frac{\lceil n/p \rceil}{2} \right\rceil + \left\lceil \frac{\lceil n/p \rceil}{3} \right\rceil + \left\lceil \frac{\lceil n/p \rceil}{5} \right\rceil + \ldots + \left\lceil \frac{\lceil n/p \rceil}{\pi_k} \right\rceil \right) \chi$$

Assume every time processor 1 finds a new prime it communicates that value to each of the other p - 1 processors in turn. If processor 1 spends λ time units each time it passes a number to another processor, its total communication time for all k primes is $k(p-1)\lambda$.

. To bring this discussion down to earth, suppose we want to count the number of primes less than 1,000,000. It turns out that there are 168 primes less than 1,000, the square root of 1,000,000. The largest of these is 997. The maximum possible execution time spent striking out primes is

$$\left(\left\lceil\frac{\left\lceil 1{,}000{,}000/p \right\rceil}{2}\right\rceil+\left\lceil\frac{\left\lceil 1{,}000{,}000/p \right\rceil}{3}\right\rceil+\ldots+\left\lceil\frac{\left\lceil 1{,}000{,}000/p \right\rceil}{997}\right\rceil\right)\chi$$

The total communication time is $168(p-1)\lambda$..

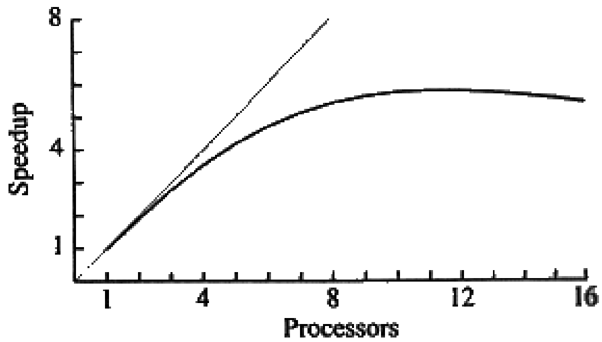


Figure 1-23 Estimated speedup of the data parallel Sieve of Eratosthenes algorithm assuming that n = 1,000,000 and $\lambda = 100\chi$. Note that speedup is graphed as a function of number of processors used. This is typical

If we know the relation between χ and λ, we can plot an estimated speedup curve for the parallel algorithm. Suppose $\lambda = 100\chi$. Figure 1-23 illustrates the estimated speedup of the data-parallel

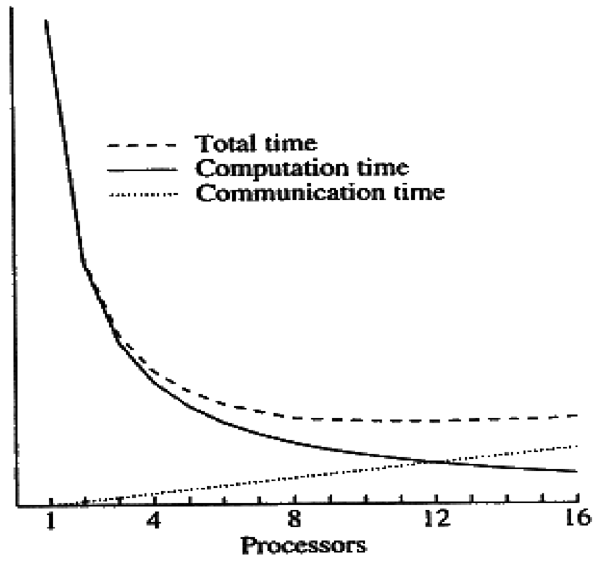


Figure 1-24 Total execution time of the data-parallel Sieve of Eratosthenes algorithm is the sum of the time spent computing and the time spent communicating. Computation time is inversely proportional to the number of processors; communication time is directly proportional to the number of processors.

algorithm.

. Notice that speedup is not directly proportional to the number of processors used. In fact, speedup is highest at 11 processors. When more processors are added, speedup declines. Figure 1-24 illustrates the estimated total execution time of the parallel algorithm along with its two components:

computation time and communication time. Computation time is inversely proportional to the number of processors used, while communication time increases linearly with the number of processors used. After 11 processors, the increase in communication time is greater than the decrease in computation time, and the total execution time begins to increase.

1.2.3.3 Pipeline approach

Figure 1-25 shows the principles of a pipeline algorithm to solve the Sieve of Eratosthenes. The first stage of the pipe generates the series of natural numbers and passes it to the next stage of the pipe. The other stages of the pipe represent the sieves. These sieve stages consider the first element of their input as prime number and use it for striking non-primes from the input stream. As a result the output stream of the stage will not contain multiples of the first prime. Finally, all the stages will contain one prime number.

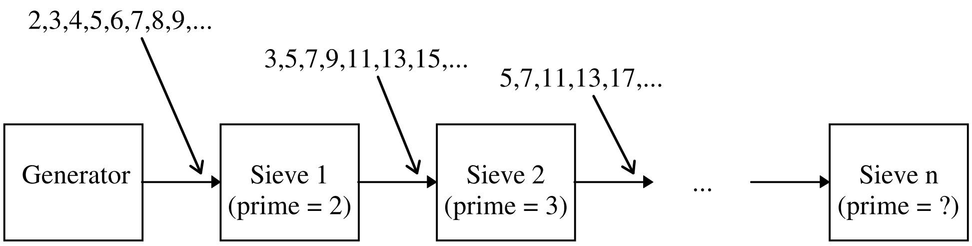


Figure 1-25 Principles of a pipeline algorithm to solve the Sieve of Eratosthenes

1.2.4 Levels of available functional parallelism

Programs written in imperative languages may embody functional parallelism at different levels, that is, at different sizes of granularity. In this respect we can identify the following four levels and corresponding granularity sizes:

- parallelism at the instruction-level (fine-grained parallelism),
- parallelism at the loop-level (middle-grained parallelism),
- parallelism at the procedure-level (middle-grained parallelism), and
- parallelism at the program-level (coarse-grained parallelism),

as shown in Figure 1-26.

Available **instruction-level parallelism** means that particular instructions of a program may be executed in parallel. To this end, instructions can be either assembly (machine-level) or high-level language instructions. Usually, instruction-level parallelism is understood at the machine-language (assembly-language) level. In addition, while considering instruction-level parallelism we will confine us to instructions expressing more or less elementary-operations, such as an instruction prescribing the addition of two scalar operands, as opposed to multi-operation instructions like instructions implying vector- or matrix-operations.

Parallelism may also be available **at the loop-level**. Here subsequent loop iterations are candidates for parallel execution. However, data dependences between subsequent loop iterations, called **recurrences**, may restrict their parallel execution. The potential speed-up is proportional to the loop limit or in case of nested loops to the product of the limits of the nested loops. Loop-level parallelism is a promising source of parallelism.

Next, there is also **parallelism** available **at the procedure-level** in form of parallel executable procedures. The extent of parallelism exposed at this level is subject mainly to the kind of the problem solution considered.

In addition, different programs (users) are obviously, independent from each other. Thus, **parallelism** is also available **at the user-level** (which we consider to be of coarse-grained parallelism). Multiple, independent users are a key source of parallelism occurring in computing scenarios. Evidently, in a problem solution different levels of parallelism are not exclusive but, they may coexist at the same time.

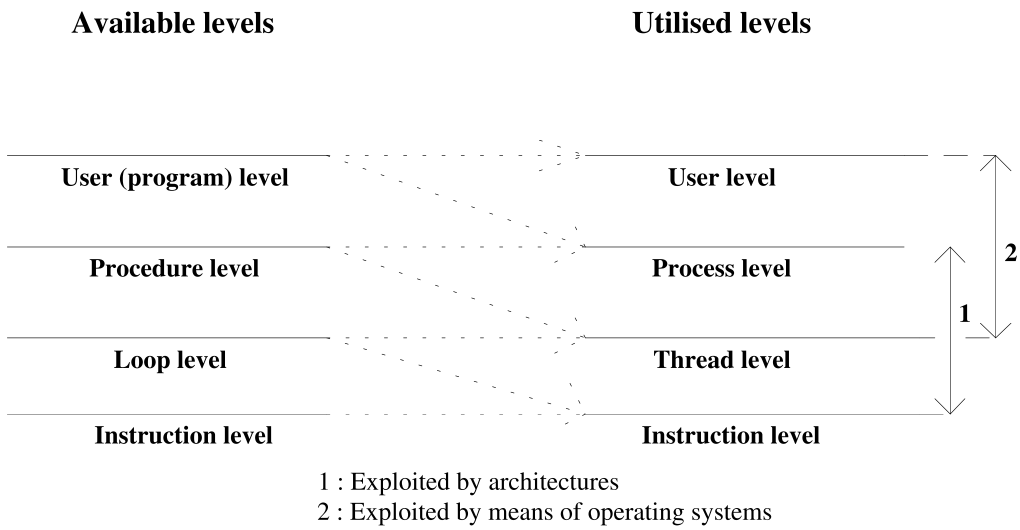


Figure 1-26. Available and utilised levels of functional parallelism

1.2.5 Utilisation of functional parallelism

Available parallelism can be utilised by architectures, compilers and operating systems conjointly for speeding up computation. Let us first discuss the utilisation of functional parallelism. In general, functional parallelism can be utilised at four different levels of granularity, that is at instruction, thread, process and user-level, (see again Figure 1-26).

It is quite natural to utilise available functional parallelism, which is inherent in a conventional sequential program, **at the instruction-level** by executing instructions in parallel. This can be achieved by means of architectures capable of parallel instruction execution. Such architectures are referred to as **instruction-level function-parallel** or simply **instruction-level parallel architectures**, commonly abbreviated as **ILP-architectures.** Since available instruction-level parallelism is typically unrevealed, i.e. implicit in traditional sequential programs, prior to execution it must be detected either by a dedicated compiler (called usually parallel optimising compiler) or by the ILP-architecture itself.

Available functional parallelism in a program can also be utilised at the **thread** and/or at the **process-level**. As discussed before, threads and processes are self-contained execution entities embodying an executable chunk of code. They are constructs to expose parallel executable pieces of code. Processes are higher-level constructs than threads, that is they expose coarser granular parallelism.

Threads and processes can be created either by the programmer using parallel languages or by operating systems that support multithreading or multitasking. They can also automatically be generated by parallel compilers during compilation of high-level language programs. Available loop- and procedure-level parallelism will be often exposed in form of threads and processes.

There are two different ways to execute threads and processes. On the one hand, they can be executed in parallel by means of specialised architectures referred to as **multithreaded** and **MIMD architectures**, respectively. Multithreaded architectures are typically specialised processors able to perform very fast context switch. The other way to execute threads and processes concurrently is the use of architectures that run threads or processes in sequence, under the supervision of a multithreaded or multitasking operating system.

In general, lower levels of available parallelism are utilised more likely directly by parallel architectures in connection with parallel optimising or parallel compilers whereas the utilisation of higher levels of parallelism relies usually more heavily on operating systems supporting concurrent or parallel execution in form of multitasking or multithreading. Figure 1-26 shows the relation between the available and utilised levels of functional parallelism.

1.2.6 Utilisation of data parallelism

Data parallelism may be utilised in two different ways. One possibility is to exploit data parallelism directly by dedicated architectures that permit parallel or pipelined operations on data elements, called data-parallel architectures (DP-architectures). The other possibility is to convert data parallelism into functional parallelism by expressing parallel executable operations on data elements in a sequential manner, by using loop constructs of an imperative language.

1.3 Classification of parallel architectures

1.3.1 Flynn's classification

Flynn's classic taxonomy [Flynn66] is based on the number of the control units as well as of the processors available in a computer. Accordingly, he introduced the notions of

- single instruction stream (i.e. the architecture has one single control unit producing one single stream of instructions), abbreviated as **SI**,
- multiple instruction streams (i.e. the architecture has multiple control units, each of them producing a distinct stream of instructions), abbreviated as **MI**,
- single data stream (i.e. one single processor is available which executes one single stream of data), abbreviated as **SD** and
- multiple data streams (i.e. multiple processors are available each of them executing a distinct stream of data), abbreviated as **MD**.

Based on these notions he categorised computers combining the possible instruction issues as well as processing aspects as follows (see Figure 1-27):

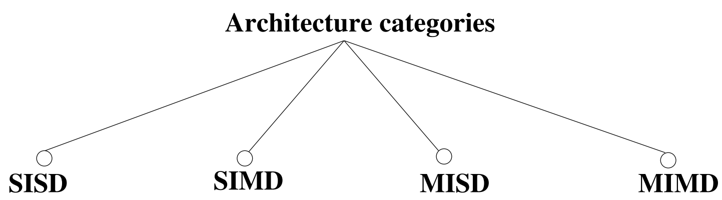


Figure 1-27 Architecture categories introduced by Flynn

Although, this is a lucid and straightforward scheme, it does not reveal or cover key aspects such as what kind of parallelism is utilised, at what level or how parallel execution is basically implemented. In the following we will present a classification scheme covering all these points of view.

1.3.2 Proposed classification

The proposed classification of parallel architectures is based on the kind of the exploited parallelism and takes into account the size of the granularity of the functional parallelism utilised, as shown in Figure 1-28.

Subject to the kind of the utilised parallelism, first, we differentiate between **data-parallel** and **function-parallel architectures**. Based on their operation principle, data parallel architectures may be subdivided into **vector, systolic, SIMD** and **associative processors**. On the other hand, function-parallel architectures may be categorised according to the granularity of the parallelism which they utilise. Thus, we can distinguish **instruction-, thread-** and **process-level parallel architectures**. Instruction-level parallel architectures are referred to as **ILP-architectures.** Thread- and process-level parallel architectures are equivalent to the class of **MIMD-architectures** as it was introduced by Flynn. However, we emphasize the special features of thread-level architectures and use the distinguishing term **multithreaded architectures** for them. Each of the major architecture classes may be subdivided into a number of subclasses.

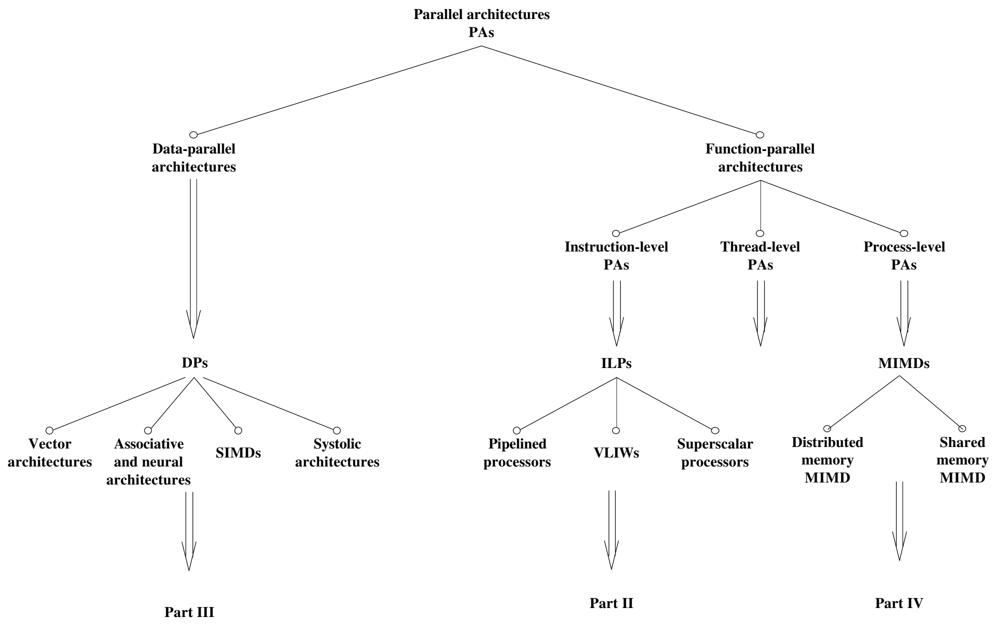


Figure 1-28. Classification of parallel architectures

According to their operation principle ILP-architectures can be classified into three major subclasses, these are the **pipelined**, **VLIW** and **superscalar processors**. Process-level architectures are subdivided into two basic classes: **multicomputers** and **multiprocessors**. They are designated also as **distributed memory computers** and **shared memory computers**, referring to the implementation of their memory system. According to the software model supported by multicomputers, they are often called **message passing machines**, too. The classification tree of parallel computer architectures is given in Figure 1-28. A separate chapter is devoted to each of the leaves of the classification tree in this, book explaining in detail their main features. In those chapters these architecture classes are further subdivided, based on the design space of these architectures.

We feel that the presented classification is a more suitable one than the well known Flynn's taxonomy, introduced in 1966. In our opinion, a recent classification has to cover aspects such as what kind of parallelism is utilised, at what level as well as how parallel execution is basically implemented, as our scheme does.

	Pipelining	**Replication**
Vector processors	+	
Systolic arrays	+	+
SIMD (array) proc.		+
Associative proc.		+
Pipelined proc.	+	
VLIW proc.		+
Superscalar proc.	+	+
Multithreaded machines	+	+
Multicomputers	+	+
Multiprocessors		+

Table 1-2. Pipelining and replication in parallel computer architectures

1.4 Parallel techniques in computer architectures

There are two basic forms of exploiting parallelism by parallel computer architectures:

1. **Pipelining**
2. **Replication**

These parallel techniques are extensively used in various parallel computer architectures as summarized in Table 1-2. In the next sections a detailed explanation of Table 1-2 is given.

1.4.1 Pipelining

In case of **pipelining** a number of functional units are employed in sequence to perform a single computation. These functional units form an assembly line or a pipeline. Each functional unit represents a certain stage of the computation and each computation should go through the whole pipeline. As far as there is a single computation to be performed, the pipeline is not able to extract any parallelism. However, when the same computation should be executed several times, these computations can be overlapped by the functional units. Assume that the pipeline consists of N functional units (stages) and the slowest requires time T to execute its function. Under such circumstances a new computation can be started at every Tth moment. The pipeline is filled up when all the functional units work on a different computation. After the pipeline is filled up, a new computation is finished at every Tth moment. The throughput of the pipeline becomes T/computation instead of the sequential throughput N*T/computation. Accordingly, a full pipeline results in speedup N if the time necessary to fill in the pipeline is negligible to the time when the pipeline is full.

Pipelining is a very powerful technique to speedup a long series of similar computations and hence, pipelining is used in many parallel architectures. It can be used inside a processor (micro-level) and among the processors or nodes (macro-level) of a parallel computer. One of the oldest class of parallel computers, the vector processor is a good example of applying the pipeline technique inside a processor. The elements of two vectors enter to the vector unit at every time step. The vector unit consisting of several functional units executes the same operation on each pair of vector elements. Modern pipelined processors and superscalar processors also use the pipelining technique to overlap various stages of the instruction execution. Processors of multithreaded architectures are usually built on the principle of pipelining, too.

Another classical example for applying pipelining can be found in systolic arrays. However, here processors of the array form the pipeline either in one or two dimensions. The wavefront array is an asynchronous version of the systolic array, where data are transferred according to the dataflow principle but the pipeline mechanism of systolic systems is preserved. An interesting and powerful application of pipelining can be found in the wormhole router of message passing computers. The introduction of the wormhole routing technique, which is a special form of pipelining, resulted in three orders of magnitude reduction in communication latency and led to a new generation of both multicomputers and multiprocessors.

Dataflow machines apply pipelining both inside and among the nodes. The dataflow nodes contain several functional units that form an instruction pipeline similarly to the pipelined processors of RISC machines. Different nodes of a dataflow machine execute different nodes of the corresponding dataflow graph. In tagged-token dataflow machines the nodes of the dataflow graph can be activated in a pipeline manner and hence the executing nodes of the dataflow machine are also used in a pipelined manner.

1.4.2 Replication

A natural way of introducing parallelism to a computer is the replication of functional units (for example, processors). Replicated functional units can execute the same operation simultaneously on as many data elements as many replicated computational recources are available. The classical example is the array processor which employs a large number of identical processors executing the same operation on different data elements. Wavefront arrays and two-dimensional systolic arrays also use the replication parallelism beside pipeline parallelism. All the MIMD architectures employ replication parallelism as their main parallel technique. Inside processor level both VLIW and superscalar processors can apply replication parallelism. Some of the multithreaded processors are also designed to exploit replication parallelism.

However, not only processor units but memory banks can also be replicated. Interleaved memory design is a well-known technique to reduce memory latency and increase performance. Similarly, I/O units can be advantageously replicated resulting in higher I/O throughput. The most trivial case of replication is the increase of address and data lines in processor buses. Microprocessor buses has developed from 8-bit buses to recent 64-bit buses and this progress will not be stopped in the near future.

1.5 Relationships between languages and parallel architectures

Though languages and parallel architectures could be considered as independent layers of a computer system, in reality, due to efficiency reasons the parallel architecture has a strong influence on the language constructs applied to exploit parallelism. In this section we show the main language concepts and their relationships with parallel architectures. In order to facilitate their comparison we apply them to solve a simple application problem.

1.5.1 Classification of parallel imperative languages

Procedural imperative languages are based on the von Neumann computational model. Concerning parallel execution schemes the von Neumann model can be further divided into three parallel programming models:

- shared memory parallel programming model
- distributed memory parallel programming model
- data parallel programming model

The shared memory parallel programming model is strongly connected with the shared memory MIMD parallel computer architecture, while the distributed memory parallel programming model supports the programming of distributed memory MIMD parallel computer architectures. Finally, the data parallel programming model is mainly related with the programming of SIMD array processors.

Low level parallel languages present the programmer with constructs closely tied to the underlying architecture. For example, Sequent C, targeted to a UMA multiprocessor, enhances C by adding constructs for declaring shared variables, forking and killing parallel processes, locking and unlocking variables to enforce mutual exclusion, and enforcing barrier synchronization. In contrast, nCUBE C, targeted to a hypercube multicomputer, enhances C by adding constructs for sending, receiving, and testing for the existence of messages between processors.

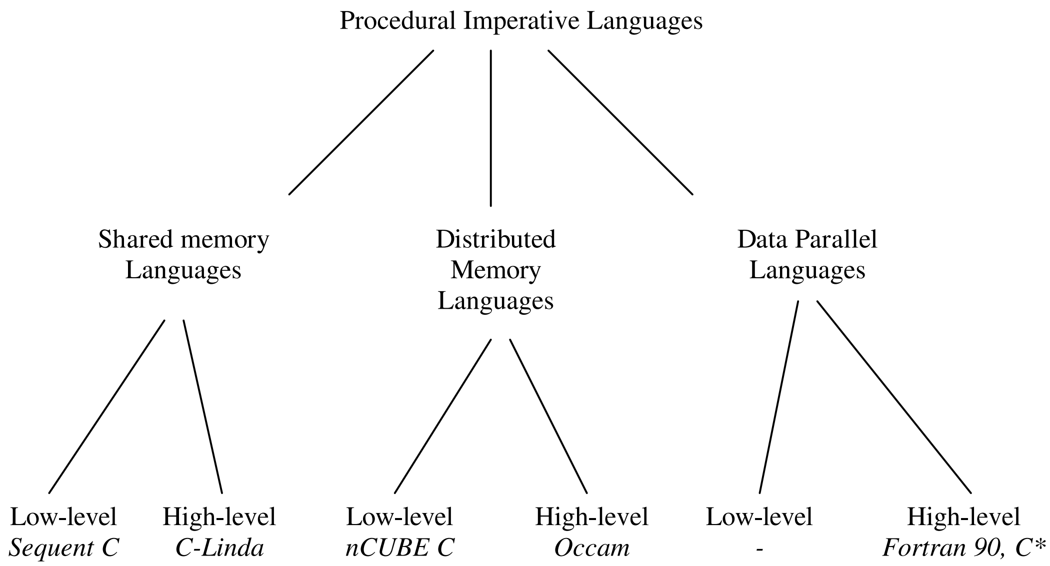


Figure 1-29 Classification of parallel imperative languages

High-level parallel languages present the programmer with a more abstract model of parallel computation. This can make programs shorter and easier to understand. Another advantage of a high-level parallel language is that it may be less dependent on the system architecture, increasing program portability. The disadvantage of a high-level language, of course, is that it makes compiler construction more challenging, since high-level languages increase the demands programmers make on the target hardware. The relationship between the models and languages are shown in Figure 1–29.

1.5.2 A simple application

We will implement a solution to a simple numerical integration problem using each language. The parallel program must compute an approximation to π by using numerical integration to find the area under the curve $4/(1 + x^2)$ between 0 and 1 (Figure 1-30).

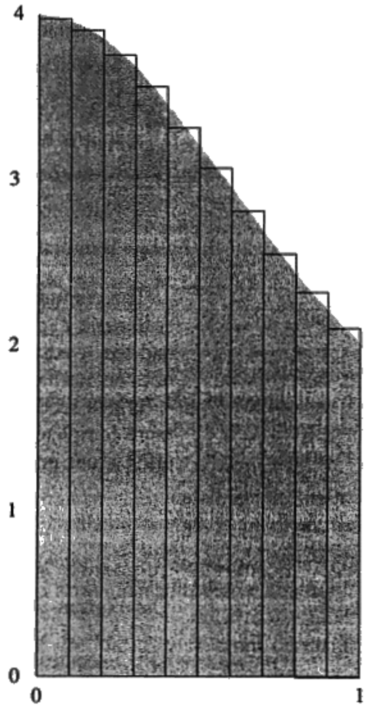


Figure 1-30 The area under the curve $4/(1 + x^2)$ between 0 and 1 is π.
The area of the rectangles approximates the area under the curve.

The interval [0, 1] is divided into n subintervals of width l/n. For each of these intervals the algorithm computes the area of a rectangle whose height is such that the curve $4/(1 + x^2)$ intersects the top of the rectangle at its midpoint. The sum of the areas of the n rectangles approximates the area under the curve. Increasing n reduces the difference between the sum of the rectangle's area and the area under the curve.

This algorithm is data-parallel, since the areas of all the rectangles can be computed simultaneously. Computing the area of each rectangle requires the same amount of work: hence load balancing is insignificant. If the language requires us to divide the work among the processors, we can do so easily.

1.5.3 Data-parallel architectures

Vector processors do not impose special language constructs, rather they require special compiler support to exploit loop parallelism related to identical operations on elements of vectors or matrices. Vectorising compilers have been successfully employed for FORTRAN programs in the field of scientific and engineering applications.

Another way to exploit loop parallelism relies on the **SPMD (Single Procedure Multiple Data) execution model**. SPMD represents a generalization of the SIMD execution model where basically only one thread of execution is applied but time to time this thread can split to N threads that work on different invocations of the same loop. When the whole loop is finished the threads are merged again into a single thread. All the N threads execute the same code similarly to the SIMD model where all processors execute identical code. However, in the SPMD model, threads can execute the same code with different speed, i.e., in a certain moment they can perform distinct instructions. Meanwhile, in the SIMD model an instruction by instruction synchronization is applied, in the SPMD model it is sufficient to synchronize the parallel threads at the end of the loop when they are merged again into a single thread. This type of synchronization is called **barrier synchronization** and its efficient implementation requires hardware support. The SPMD execution model is typically employed on MIMD parallel architectures where processors are able to execute distinct instructions.

Similarly to the vector machines, neither systolic arrays nor associative processors require language support. The only data parallel architecture that is sensitive to languages is the SIMD (array processor) machine. Data parallel languages developed for SIMD computers typically contain parallel data structures like 1-, 2- and more dimensional arrays that are handled as single objects. It means that an operation on such an object is performed in parallel at each element of the object without applying the usual loop constructs. For example, to add two 2-dimensional arrays requires two nested loops in FORTRAN, while it is programmed by a single statement in DAP FORTRAN as shown in Figure 1-31.

FORTRAN code	**DAP FORTRAN code**
`do 2 j=1,n`	
`do 1 i=1,n`	
`C(i,j) = A(i,j) + B(i,j)`	`C = A + B`
`1 continue`	
`2 continue`	

Figure 1-31 Programming sequential and parallel data structures

There is only a single common control unit for all the processors in SIMD machines and hence, each processor executes the same code. However, there are situations when only a subset of the elements of an array should be involved in an operation. A new language construct, called the mask, is needed to define the relevant subset of arrays. The mask technique is a crucial part of each data parallel languages. Again a DAP FORTRAN example illustrates the use of masks compared to conventional FORTRAN programs in Figure 1-32. Both programs assign value zero to those elements of the 2-dimensional array A that are less than 3. In FORTRAN, this simple operation is realised by two nested loops. In DAP FORTRAN, a single assignment is sufficient to define the operation. The mask on the left hand side of the assignment makes it sure that only the chosen elements of the array A will be updated.

FORTRAN code	**DAP FORTRAN code**
`do 2 j=1,n`	
`do 1 i=1,n`	
`if (A(i,j).lt.3.0) A(i,j) = 0.0`	`A(A.lt.3.0) = 0.0`
`1 continue`	
`2 continue`	

Figure 1-32. Mask technique in DAP FORTRAN

Finally, data parallel languages contain language constructs to specify the allocation of processors for the elements of the parallel data structures. While the application of parallel data structures and masks simplifies and shortens the program text, the allocation constructs lead to the expansion of programs. All the three new techniques require a special way of thinking significantly different from conventional FORTRAN and that is one of the main reasons why SIMD machines could not gain real popularity.

1.5.3.1 FORTRAN 90

In 1978 the ANSI-accredited technical committee, X3J3, began working on a new version of the FORTRAN language. In the early 1990s the resulting language, Fortran 90, was adopted as an ISO and ANSI standard. Fortran 90 is a superset of FORTRAN 77 and shows many similarities with DAP FORTRAN. It includes all the features of FORTRAN 77, plus many new features, some of them are listed below:

- array operations
- Improved facilities for numerical computations
- User defined data types, structures, and pointers
- Dynamic storage allocation
- Modules to support abstract data types
- Internal procedures and recursive procedures

The committee also marked many language features as obsolete, including arithmetic IF, some Do construct variations, assigned GO TO, assigned formats, and the H edit descriptor.

1.5.3.1.1 Fortran 90 programmer´s model

The Fortran 90 programmer has a model of parallel computation similar to a PRAM (Figure 1–33). A CPU and a vector unit share a single memory. The CPU executes sequential instructions, accessing variables stored in the shared memory. To execute parallel operations, the CPU controls the vector unit, which also stores and fetches data to and from the shared memory.

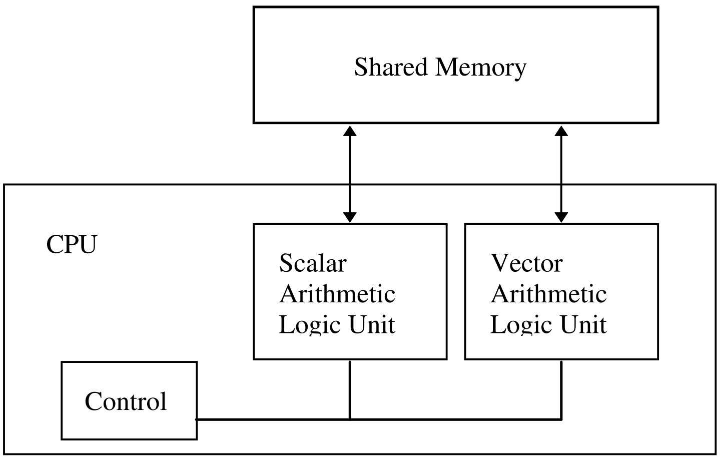


Figure 1-33 The Fortran 90 programmer's model of parallel computation

1.5.3.1.2 Sample programs

Listing 1–1 contains a Fortran 90 program to compute π using numerical integration. The parameter N, declared in line 1, is the number of subintervals. In line 2 we declare parameter LONG, used to set up floating-point variables with at least 13 digits of precision and exponents covering at least the range 10^{-99} through 10^{99}. These high precision floating-point variables are declared in lines 3 and 5. In line 6 we compute the width of each subinterval. The array ID, declared in line 4 and initialised in line 7, represents the subinterval number associated with each array element. In line 8 we compute in parallel the midpoint of each of the subintervals, and in line 9 we compute in parallel the height of the function curve at each of these points. Line 10 contains the call to the SUM function, which adds the heights of all of the-rectangles, then multiplies the total height by the rectangles' width to yield the area under the curve. In line 13 we print the result.

```
1.      INTEGER, PARAMETER :: N = 131072
2.      INTEGER, PARAMETER :: LONG = SELECTED_REAL_KIND(13,99)
3.      REAL(KIND=LONG) PI, WIDTH
4.      INTEGER, DIMENSION(N) :: ID
5.      REAL(KIND=LONG), DIMENSION(N) :: X, Y
6.      WIDTH = 1.0-LONG / N
7.      ID = (/ (I, I = 1, N) /)
8.      X = (ID - 0.5) * WIDTH
9.      Y = 4.0 / (1.0 + X * X)
10.     PI = SUM(Y) * WIDTH
11. 10 FORMAT (' ESTIMATION OF PI WITH ',I6, &
12.     'INTERVALS IS ',F14.12)
13.     PRINT 10, N, PI
14.     END
```

Listing 1-1 Fortran 90 program to compute π using numerical integration. The line numbers are for reference purposes only; they are not part of the program.

*1.5.3.2 C**

C* is a data-parallel extension of the C language suitable for programming the Connection Machine a typical SIMD parallel architecture.

1.5.3.2.1 Programmer's model

C* programmers imagine they are programming a SIMD computer consisting of a frontend uniprocessor attached to an adaptable back-end parallel processor (Figure 1–34). The front-end processor stores the sequential variables and executes the sequential code. The back-end processors store the parallel variables and execute the parallel portions of the program. Each processing element in the back end has its own local memory. There is a single flow of control; at any one time either the front-end processor is executing a sequential operation, or the back-end processors are executing a parallel operation.

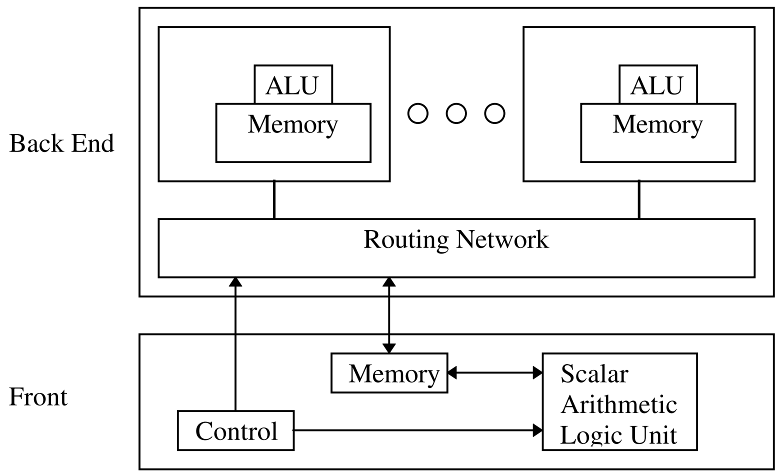


Figure 1-34 C programmer's model of parallel comutation*

The back-end processor array is adaptable—that is, programmers can select the size and shape of the processing elements they want to activate. These parameters are independent of the size and topology of the underlying physical machine. For this reason we sometimes refer to processing elements as virtual processors. Furthermore, the configuration of the back-end processor array is adaptable between different points in the same program.

1.5.3.2.2 Sample program

Listing 1–2 illustrates how to implement our numerical integration program in C*. The program is written for a Connection Machine with 8192 processing elements. In line 1 we define INTERVALS so that the number of rectangles is a multiple of the number of processing elements. The C* programmer thinks in terms of virtual processors, because the total number of elements in a shape can exceed the number of processing elements in the target machine. The compiler and operating system take care of virtual processor emulation.

In line 2 we declare a one-dimensional shape that corresponds to the set of subintervals into which we have divided the line segment [0, 1]. In lines 4 and 5 we declare two scalar double-precision variables. Line 6 is the start of the *with* statement, which puts us in a parallel context. We declare a double precision variable *x* of shape type *span* in line 7. There is one value of *x* for each element of the shape. In line 8 we compute the midpoint of each rectangle on the X axis. Note the use of function *pcoord*, which returns a unique value in the range 0...131071 to each of the elements of *x*. In line 9 we compute the height of the function curve at each of the midpoints, add these heights, and store the result in scalar variable *sum*. We multiply the total height by the rectangles' width in line 11 to determine the total area, and print the result in line 12.

```
1.   #define INTERVALS (8192 * 16)
2.   shape [INTERVALS] span;
3.   main () {
4.   double sum;                          /* Sum of areas */
5.   double width=1.0/INTERVALS;          /* Width of rectangle*/
6.   with(span) {
7.     double:span x;      /*Midpoint of rectangle on x axis*/
8.     x = (pcoord(0) + 0.5) * width;
9.     sum = (+= (4.0/(1.0+x*x)));
10.  }
11.  sum *= width;
12.  printf ("Estimation of pi is %14.12f\n", sum) ;
13.  }
```

Listing 1-2 Pi computation program written in C, version 6.0. The line numbers are for reference purposes only; they are not part of the program.*

1.5.4 Instruction- and thread-level function-parallel architectures

ILP architectures are not supported by special language constructs. Intelligent optimising compilers are necessary to exploit that level of parallelism.

A particularly efficient way of programming multithreaded machines relies on the data driven execution model and applicative programming languages. In this case there is no need of any explicit specification of parallelism, compilers are easily able to exploit inherent parallelism from the program text. However, coarse-grain multithreaded architectures are usually programmed similarly to process-level MIMD machines either by concurrent or parallel languages.

1.5.5 Process-level function-parallel architectures

Beyond the creation and termination of threads concurrent and parallel languages provide tools for communicating and synchronizing these threads. Figure 1-35 illustrates the progress of synchronizing tools from low level to high level constructs. The two main progress lines correspond to the shared memory and distributed memory MIMD machines.

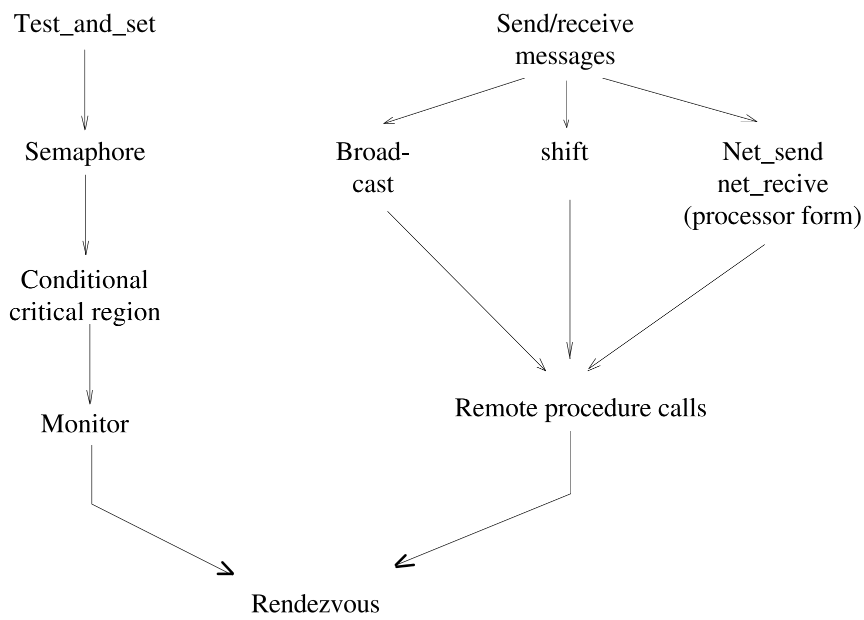


Figure 1-35. Progress of language constructs used for syncronization

1.5.5.1 Shared memory MIMD architectures

Processes and threads in shared memory MIMD computers use shared data structures to communicate. Simultaneous access of shared data structures sometimes should be prohibitive and hence, they require synchronization. Obviously, a shared data structure must not be read meanwhile it is updated. The writer and reader process should access the shared data structure in mutual exclusive way. This problem, called **mutual exclusion**, is one of the many various synchronization problems. For such syncronizations the lowest level (machine instruction level) language tool is the **test_and_set operation** which should be implemented by hardware on any machine.

Based on the test_and_set mechanism various **semaphores** and semaphore operations (P and V) can be defined [Dijkstra68]. The two most well known versions are the **binary semaphore** and the **queue-semaphore**. The mutual exclusion problem and its solution by a binary semaphore is shown in Figure 1-36.

The main difference between the two types of semaphores comes from the scheduling mechanism behind the P-operation. In case of binary semaphores the test phase of the P-operation is executed on a busy-wait way. The process keeps the processor busy as long as the requested semaphore is locked. Queue-semaphores enable to suspend the process that executes a P-operation on a locked semaphore. When the semaphore is opened by another process executing a V-operation, the suspended process can acquire the shared resource and becomes ready-to-run.

Programming process communication by semaphores is very error-prone and hence, several higher-level synchronization constructs have been proposed and introduced into concurrent and parallel languages. The two most notable ones are the conditional critical region [BHansen73] which was applied in the language Edison [BHansen81] and the monitor [Hoare74] which appeared, for example, in Concurrent Pascal [BHansen75]. Another approach is the Linda concept that hides low-level synchronisation details into simple shared memory access primitives.

1.5.5.1.1 C-LINDA

Linda consists of several operations that work on tuple space, a shared associative memory. Incorporating Linda operations into a sequential base language yields a parallel programming language. For example, in C-Linda, Linda primitives have been added to C. In this section we describe the Linda programming model and present two C-Linda programs to solve the numerical integration problem.

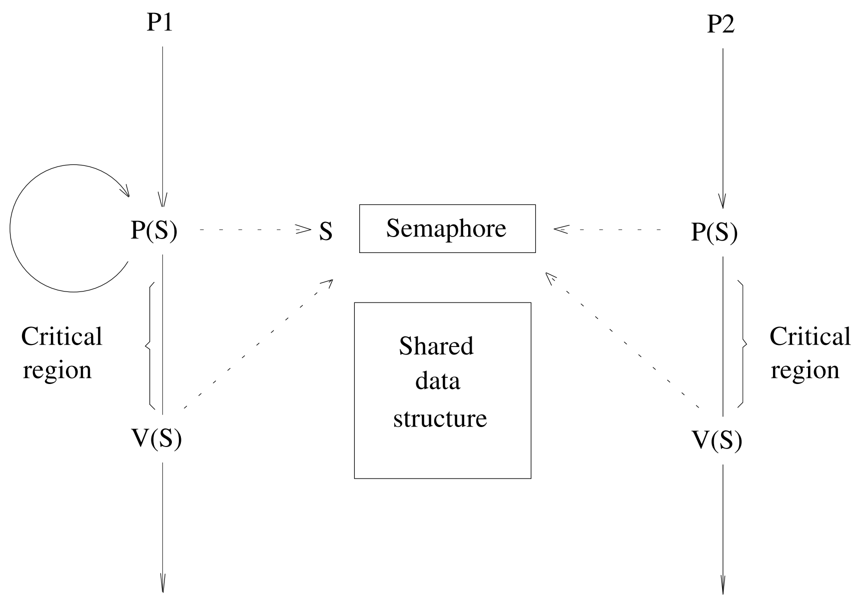


Figure 1-36. Using semaphore to solve the mutual exclusion problem

1.5.5.1.1.1 Programmer's Model

Linda is a MIMD model of parallel computation. The Linda programmer envisions an asynchronously executing group of processes that interact by means of a associative shared memory, tuple space. Tuple space consists of a collection of logical tuples. Parallelism is achieved by creating process tuples, which are evaluated by processors needing work. Parallel processes interact by sharing data tuples (see Figure 1–37). After a process tuple has finished execution, it returns to tuple space as a data tuple.

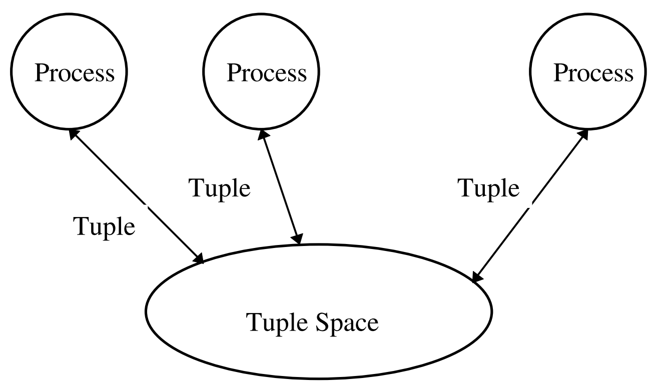


Figure 1-37 The Linda programmer's model of parallel computation

In many MIMD languages, such as Occam, processes may interact with each other in complicated ways (see Figure 1–40). Proponents of Linda say that forcing all process interactions to occur through tuple space simplifies parallel programming.

1.5.5.1.1.2 C-Linda Language Constructs

Six functions enable a process to interact with tuple space.

Function out, passed tuple t as an argument, causes t to be added to tuple space. The process executing the out operation continues immediately.

Function in, passed template s, removes from tuple space a tuple t that matches template s. The values of the actual parameters in t are assigned to the formal parameters in s. The in operation blocks the process until a suitable matching tuple is found. If multiple tuples match, an arbitrary one is chosen.

Function rd, passed template s, behaves in the same way as function in, except that the matching tuple is not removed from tuple space.

Function eval, passed tuple t, forks a new process to evaluate t. Once t has been evaluated, it is placed in tuple space as an ordinary data tuple.

Function inp is a nonblocking version of in. It returns a 1 if a matching tuple is retrieved, 0 if not.

Function rdp is a nonblocking version of rd. It returns a 1 if a matching tuple is found, 0 if not.

1.5.5.1.1.3 Sample Programs

We will examine two C-Linda programs to solve the π computation problem. One program implements a **master/worker algorithm**; the other implements a **divide-and-conquer algorithm**.

The first C-Linda program appears in Listing 1–3. Each process will execute function *work* (lines 3-16) to compute the area of its share of the rectangles.

Function *main*, the starting point for C programs, is reserved by the runtime system to start up C-Linda. Function *real_main* is the user function where execution begins. In line 21 we compute len, the maximum number of rectangles assigned to any process. Variable *left_overs*, computed in line 22, is the number of processes that will have to compute *len* intervals; the rest will compute *len-l* intervals.

The *for* loop in lines 23-26 spawns NUM_WORKERS-1 processes. Each process will execute function *work*, computing the sum of the heights of its share of rectangles, then returning the total area. When the process executes the return statement, it inserts into tuple space a tuple whose first field is "worker" and whose second field is the area. Then it terminates.

After forking off NUM_WORKERS-1 parallel processes in lines 23-26, the parent process executes function *work* itself. It assigns the value returned from the function to variable *sum*. In lines 28-31 the parent process collects from tuple space the subtotals generated by the other processes and adds each subtotal to *sum*. When it has added all these values, it prints the answer in line 32.

```
                    /*  Pi computation in C-Linda */

/*  Compute the area under the curve 4/(1 + x*x)
    between 0 and 1 */
1.  #define INTERVALS 4000000
2.  #define NUMLWORKERS 7
3.  double work (start, finish, width)
4.  int start, finish;
5.  double width;        /* Width of interval */
6.  {
7.     int i;
8.     double sum;      /* Sum of areas */
9.     double x;        /* Midpoint of rectangle on x axis*/
10. sum = 0.0;
11.    for (i = start; i < finish; i ++) {
12.         x = ( i+0.5 ) *width;
13.         sum += 4.0/ ( 1.0 + x*x);
14.    }
15.    return (sum * width);
16. }

17. real_main() {
19.    int i, left_overs, len, w;
20.    double result, sum;
21.    len = INTERVALS/NUM_WORKERS + l;
22. left_overs = lNTERVALS%NUM_WORKERS;
```

```
    /* This process will do some work too, so start the worker
       count/id variable, w, at 1 rather than at 0. */
       for (   i = 0, w = 1; w < NUM_WORKERS;
               i += len, ++w) {
           if (i == left_overs) --len;
       eval("worker", work(i, i+len, 1.0/INTERVALS));
       }
       sum = work(i, INTERVALS, 1.0/INTERVALS);
       for (w = 1; w < NUM_WORKERS; ++w) {
           in ("worker", ? result ) ;
           sum += result ;
       }
       printf ("Estimation of pi is %14.12f\n", sum) ;
    }
```

Listing 1-3 Master/worker-style C-Linda program to compute π. The line numbers are not part of the program; they are for reference puroses only.

The second C-Linda program, listed in Listing 1–4, implements a divide-and-conquer algorithm to solve the π computation problem. In this version of the program the original process, executing *real_main*, calls function *work* with arguments specifying the entire set of intervals.

Function *work* implements the parallel divide-and-conquer algorithm. If the number of intervals specified in the formal parameters to work is greater than INTERVAL_LIM, the process executing *work* uses the *eval* primitive to fork off two parallel processes, each of which calls *work* recursively to compute the area under the curve for half the intervals. The forking process retrieves the results of the computations, sums them, and returns a data packet containing the total area to tuple space. If the number of intervals is less than or equal to INTERVAL_LIM, the process computes the area under the curve for these intervals and returns a data packet containing the area to tuple space.

```
     /* Pi computation in C-Linda: divide and conquer approach. */

/* Compute the area under the curve 4/(1 + x*x) between 0 and 1 */

#define INTERVAL_LIM      50000
#define INTERVALS   400000

double work (id, start, finish, width)
    int id, start, finish;
    double width;                /* Width of interval */
{
    int i;
    int length = finish - start;
    double sum, sum2;
    double x;

    if (length > INTERVAL_LIM) {
      eval(   "worker", id, work(id< < 1, start,
              start+(length > > 1), width) );
      eval(   "worker", id, work((id< < 1)+1,
             start+(length >> 1), finish, width) );
      in("worker", id, ?sum);
      in("worker", id, ?sum2);
      return (sum+sum2);
    }
    for (i = start; i < finish; i ++) {
      x = ( i+0.5 ) *width;
      sum += 4.0/ ( 1.0 + x*x);
    }
    return (sum * width);
}
```

```
real main() {
    printf("Estimation of pi is %14.12f\n",
    work(1, 0, INTERVALS, 1.0/INTERVALS) );
}
```

Listing 1-4 Divide-and-conquer-style C-Linda program to compute π.

1.5.5.2 Distributed memory MIMD architectures

Languages to program distributed memory architectures use message passing operations like *send* and *receive* to specify communication and synchronization among processors. Similarly to the semaphore operations they represent very low-level and error-prone operations and hence, higher-level language constructs like broadcast, shift, net_send, and net-receive have been introduced. The net_send and net_receive construct appear in 3L Parallel C and they support the processor farm programming technique. A further step in safe programming of distributed memory architectures is the introduction of remote procedure calls (RPCs). Application of RPCs significantly reduces the main problem of message passing architectures, namely the creation of deadlock situation where processors mutually wait for one another. For example, Occam-3 introduced the RPC concept in order to make transputer programming safe.

The two main lines of synchronization tools lead to the rendezvous language construct that combines the features of remote procedure calls and monitors. The rendezvous concept could be implemented on both shared and distributed memory machines. The most well-known language employing the rendezvous concept is Ada.

Recently, many efforts have been devoted to avoid machine dependencies and to define parallel interfaces that can be easily ported to various MIMD type parallel computer architectures. The two most famous and celebrated interfaces are:

1. PVM (Parallel Virtual Machine) [Geist94]
2. MPI (Message Passing Interface) [GropLusk94]

Both are based on the message passing programming paradigm and provide a rich library of send/receive type commands. Additionally, MPI realizes many high level communication constructs that simplify the programming of MIMD computers. PVM is intended to support first of all workstation clusters, while MPI was designed as a standard programming interface for massively parallel (both distributed memory and distributed shared memory) computers.

In the next two sections we show a low-level (nCUBE C) and a high-level message passing language (Occam-2).

1.5.5.2.1 nCUBE C

1.5.5.2.1.1 The Run-Time Model

An nCUBE programmer usually writes a single program that executes on every node processor. Each node program begins execution as soon as the operating system loads it into the node. To implement a data-parallel application, the programmer assigns each processor responsibility for storing and manipulating its share of the data structure. This is called programming in the SPMD (Single Program, Multiple Data) style [Karp87]. Processors work on their own local data until they reach a point in the computation when they need to interact with other processors to: swap data items, communicate results, perform a combining operation on partial results, etc. If a processor initiates a communication with a partner that has not finished its own computation, then the initiator must wait for its partner to catch up. Once the processors have performed the necessary communications, they can resume work on local data. One way to view the entire computation is to consider the activities of the processors, as they cycle between computing and communicating, with occasional periods of waiting (Figure 1–38).

A SPMD program executing on a multicomputer such as the nCUBE consists of alternating segments in which processors work independently on local data, then exchange values with other processors through calls to communication routines. In Figure 1–38 black represents computing, white represents waiting, and grey represents communicating.

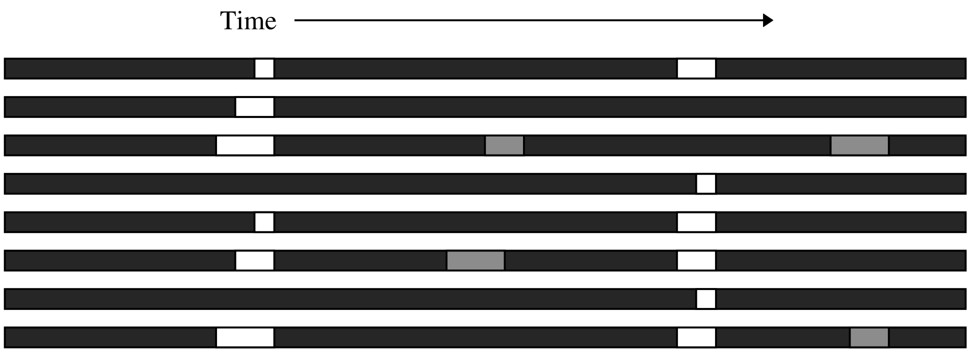


Figure 1-38 This figure gives an abstract view of the nCUBE C run-time model. Every processor is represented by a horizontal line. Black represents computing on the processor, white represents time spent on communicating with other processors, and grey shows time spent waiting for a message.

1.5.5.2.1.2 Extensions to the C Language

The C programs running on the nodes are inherently parallel, in the sense that every node is active from the time its program is loaded. For this reason there is no need for a process creation mechanism in nCUBE C. In fact, three new functions are sufficient for us to run our sample application on the nCUBE:

whoami: Returns information about the node process, including the dimension of the allocated hypercube and the position of the node in the allocated hypercube.

pwrite: Sends a message to another node. Arguments to this function include the destination node number, the address of the message to be sent, the length of the message, and the message type, an integer. Function nwrite is nonblocking.

nread: Receives a message from another node. Arguments to this function include the source node number, the message type, the address where the message is to be stored, and the maximum number of bytes to be read. The value returned from the function is the number of bytes actually read. The user has the option of specifying that the message to be read can be of any type or from any source, or both. Function nread blocks until a message fitting the source and type criteria has been read.

1.5.5.2.1.3 Sample Program

An nCUBE C program to perform the n computation appears in two parts (Listing 1–5.a and 5.b). Look at the code in Listing 1–5.a. The two "include" files allow the node program to call functions from the C I/O library. All messages sent by the fan-in function are of type FANIN.

```
/* Node program */
#include < special.h>
#include < stdio.h>

main (argc, argv)
     int argc; char *argv[];
{
     void fan_in();
     int cube_dim;        /* Dimension of allocated hypercube*/
     int i;
     int intervals;       /* Number of intervals * /
     int nodenum;         /* Position of node in allocated hypercube*/
     int not_used;        /* Place holder */
     int num_procs;       /* Number of active processors */
     double sum;          /* Sum of areas */
     double width;        /* Width of interval */
     double x;            /* Midpoint of rectangle on x axis */

     /* argv[0] is name of file containing host program */
```

```
    /* argv[l] is name of file containing node program */
    /* argv[2] is dimensional of allocated hypercube */
    /* argv[3] is number of intervals */

    intervals = atoi ( argv [ 3 ] );
    width = 1.0 / intervals;
    whoami (&nodenum, &not_used, &not_used, &cube_dim);
    num_procs = 1 < < cube_dim;
    sum = 0.0;
    for (i = nodenum; i < intervals; i += num_procs) {
       x = ( i+0.5 ) *width;
       sum += width* ( 4.0/ ( 1.0 + x*x) );
    }
    fan_in (&sum, nodenum, cube_dim);

    if( !nodenum )
       printf ('Estimation of pi is %14.12f\n", sum);
}
```

Listing 1-5.a First part of nCUBE C program to compute π using numerical integration.

Each node processor executes the function *main*. Every processor has its own local copy of each of the variables declared inside main. The nodes will estimate n by approximating the area under the curve $4/(1 + x^2)$ between 0 and 1 with *intervals* rectangles. The node process calls function *whoami* to determine the dimension of the allocated hypercube, as well as its position in the allocated hypercube. The number of processors is 2 raised to the power of the cube dimension.

Once a processor has computed the sum of the areas of its share of the rectangles, it participates in calling function *fan_in*. After completion of function *fan_in*, processor 0 has the total value, which it prints.

```
void fan_in (value, nodenum, cube_dim)
    double *value; int nodenum; cube_dim;
{
    int dest, i, source, type;
    double tmp;
    type = FANIN;
    for (i = cube_dim-1; i >= 0; i- -)
        if (nodenum < (1 << i) ) {
            source = nodenum ^ (1 << i);
            nread (&tmp, sizeof (double), &source, &type);
            *value += tmp;
        }else if (nodenum < (1 << i (i+1) ) ) {
            dest = nodenum ^ (1 << i );
            nwrite (value, sizeof (double), dest, type);
        }
}
```

Listing 1-5.b nCUBE C source code for function fan_in, which accumulates each node's subtotal into a sum stored on node 0.

Function *fan_in* is shown separately in Listing 1-5.b. Passed the address of a double-precision floating-point value, the node number, and the cube dimension, function *fan_in* computes the sum of every processor's value at that address, using a binomial tree communication pattern (Figure 1–39). Initially all the processors are active. During each iteration the algorithm divides the active processors into two sets of equal size. The processors in the upper half send their values to the processors in the lower half. The processors in the lower half add the two values, while the processors in the upper half become inactive. If a processor is sending a value, variable *dest* contains the number of the processor receiving the value; if a processor is receiving a value, variable *source* contains the number of the processor sending the value. After one iteration per cube dimension, processor 0 is the only remaining active processor, and it contains the global sum in variable *value*. Other processors will have either the original value or a partial sum.

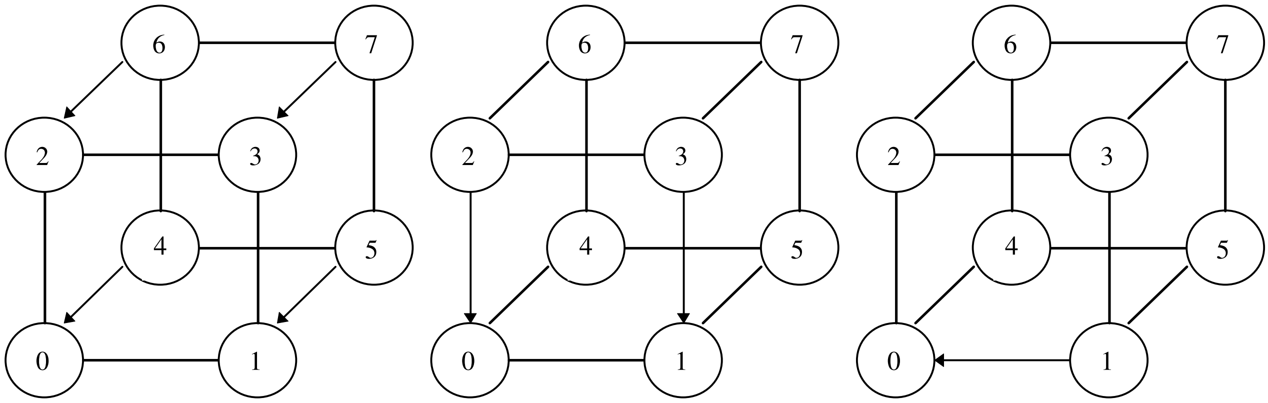


Figure 1-39 Function fan_in in Listing 1-5.b computes the sum of values stored on the hypercube nodes using the communication pattern shown by the arrows. Together, the arrows form a binomial tree.

1.5.5.2.2 OCCAM-2

Inmos Limited of Great Britain developed Occam-2 as a programming language for its Transputer series of processors. The design of Occam-2 was heavily influenced by the work done by Hoare on communicating sequential processes [Hoare78]. Because the Occam-2 language has been closely associated with the evolution of the Transputer chip, it has improved as the hardware of successive Transputers has become more sophisticated. For example, the original Occam language did not use floating-point data, because the T414 processor did not support it. This feature was added to the language with the introduction of the T800 processor, which used floating-point hardware.

1.5.5.2.2.1 Programmer´s Model

At the macro level the Occam-2 programmer views a parallel computation as a collection of asynchronously executing processes communicating via a synchronous message-passing protocol. An Occam-2 program is built from three kinds of primitive process: assignment, input, and output. The programmer can assemble more complicated processes by specifying when they must execute sequentially and when they can execute in parallel.

Figure 1–40 represents the structure of a hypothetical Occam-2 program. At the global level the program is one Occam-2 process, indicated by the ellipse E around the graph. In this example, the process has six subprocesses that must execute sequentially. (Control dependencies are shown with dashed arrows.) All but the fourth process are primitive processes, indicated by small circles. The fourth process is a built-up process, as indicated by ellipse D. Built-up processes can represent either a sequential or a parallel collection of simpler processes. The primitive process represented by a filled grey circle is an input process

inside a sequential collection of processes (A)
inside a parallel collection of processes (B)
inside a sequential collection of processes (C)
inside a parallel collection of processes (D)
inside a sequential collection of processes (E)

The assignment process, one of the three primitive processes in an Occam-2 program, assigns the value of an expression to a variable. An input process assigns to a variable the value read from a channel. An output process evaluates an expression and writes the result to a channel. A **channel** is a one-way, point-to-point, synchronous link from one process to another process. An Occam-2 channel is a logical entity independent of the underlying hardware.

1.5.5.2.2.2 Sample Program

The Occam-2 program in Listing 1–6 computes π using the numerical integration method already described. Lines 1–3 define constants. *N* is the number of intervals, *PROCESSES* is the number of processes we will create, and *CHUNK* is the number of intervals per process. (For the program to execute correctly, *N* must be a multiple of *PROCESSES*.) In line 4 we define one channel per process. The process will use this channel to output its partial sum to the process that is finding the grand total.

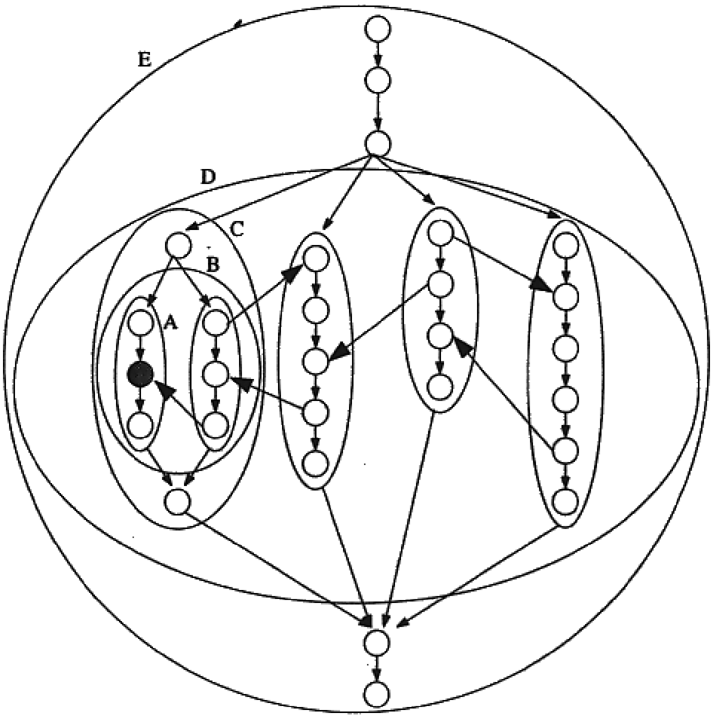


Figure 1-40 The Occam-2 programmers model of parallel computation. Primitive processes appear as circles. Ellipses indicate processes built up as sequential or parallel collections of processes. Control dependencies are shown as dashed arrows. Communications appear as bold arrows between primitive processes.

Lines 5-31 constitute a parallel construct with processes. All but one of the processes execute lines 6-17. Each process computes the sum of the areas of its share of rectangles. The last process executes lines 18-31 where it collects the output of the sum-finding processes, computes the grand total, and prints the result.

```
1. DEF N = 400000 :
2. DEF PROCESSES = 8 :
3. DEF CHUNK = N / PROCESSES :
4. CHAN sum [PROCESSES] :
5. PAR
6.   PAR i = [0 FOR PROCESSES]
7.   REAL64 x, localsum, width :
8.   SEQ
9.     localsum := 0.0
10.     width := 1.0 / N
11.     x := ((i * CHUNK) + 0.5) * width
12.     SEQ i = [0 FOR CHUNK]
13.       SEQ
14.         localsum := localsum + (4.0 / (1.0 + (x * x)))
15.         x := x + width
16.       localsum := localsum * width
17.       sum[i] ! localsum
18.     REAL64 pi :
19.     INT got[PROCESSES] :
20.     SEQ
21.       pi := 0.0
22.       SEQ i = [0 FOR PROCESSES]
23.         got[i] := FALSE
24.       SEQ i = [0 FOR PROCESSES]
25.         REAL64 y :
```

```
        SEQ
          ALT i = [0 FOR PROCESSES]
            (got[i] = FALSE) & sum[i] ? y
            got[i] := TRUE
          pi :=pi +y
    output ! "Approximation to pi is " ; pi
```

Listing 1-6 An Occam-2 program to compute π using numerical integration. The line numbers are not part of the program, they are for reference purposes only.

1.5.6 Summary

Table 1-3 summarizes the relationship among computational models, languages, explicit language constructs and parallel computer architectures. A very thorough study of concurrent programming languages and techniques can be found in [Andrews91].

Specification of parallelism	Execution model	Language	Parallel architecture
Loops	Vectorization	Conventional procedural	Vector computers
Loops	SPMD	Conventional procedural	MIMD architectures with barrier synchr.
1 Explicit declaration of parallel data structures 2. Explicit allocation of parallel data structures to processors	SIMD	Data parallel	SIMD (array processors)
No explicit specification	Instruction-level functional parallel	Any	ILP architectures
No explicit specification	Data driven	Applicative	Multithreaded architectures
1. Explicit partitioning of program into parallel processes 2. Explicit synchroniza-tion among processes	Processes communicating through shared data	Concurrent	Shared memory MIMD machines
1. Explicit partitioning of program into parallel processes 2. Explicit messages among processes 3. Explicit allocation of processes to processors	Processes communicating by message passing	Parallel	Distributed memory MIMD machines

Table 1-3. Summary of forms of parallelism

1.6 Speedup, Amdahl's law

Two important measures of the quality of parallel algorithms implemented on multiprocessors and multicomputers are speedup and efficiency. The **speedup** achieved by a parallel algorithm running on p processors is the ratio between the time taken by that parallel computer executing the fastest serial algorithm (on one processor) and the time taken by the same parallel computer executing the corresponding parallel algorithm using p processors. The **efficiency** of a parallel algorithm running on p processors is the speedup divided by p.

An example illustrates the terminology. If the best known sequential algorithm executes in 8 seconds on a parallel computer, while a parallel algorithm solving the same problem executes in 2 seconds when five processors are used, then we say that the parallel algorithm "exhibits a speedup of 4 with five processors." A parallel algorithm that exhibits a speedup of 4 with five processors "has an efficiency of 0.8 with five processors."

Some define the speedup of a parallel algorithm running on p processors to be the time taken by the parallel algorithm on one processor divided by the time taken by the parallel algorithm using p processors. This definition can be misleading since parallel algorithms frequently contain extra operations to facilitate parallelization. Comparing the execution time of a parallel algorithm running on many processors with that same algorithm running on one processor can exaggerate the speedup, because it masks the overhead of the parallel algorithm. In this course we use the term **parallelizability** to refer to the ratio between the time taken by a parallel computer executing a parallel algorithm on one processor and the time taken by the same parallel computer executing the same parallel algorithm on p processors.

1.6.1 Superlinear speedup

Is superlinear speedup possible? In other words, is it possible for the speedup achieved by a parallel algorithm to be greater than the number of processors used? The answer is that it depends upon the assumptions made. Some people argue that speedup cannot be greater than linear (e.g., see [Faber86]). They base their proof on the premise that a single processor can always emulate parallel processors. Suppose a parallel algorithm A solves an instance of problem Π in Tp units of time on a parallel computer with p processors. Then algorithm A can solve the same problem instance in $p \times Tp$ units of time on the same computer with one processor through time slicing. Hence the speedup cannot be greater than p. Since parallel algorithms usually have associated overhead, it is most likely there exists a sequential algorithm that solves the problem instance in less than $p \times Tp$ units of time, which means the speedup would be even less than linear.

We disagree with **two assumptions** made in the previous proof. One assumption is **algorithmic**, and the other is **architectural**. Let us first examine the questionable algorithmic assumption.

Is it reasonable to choose the algorithm after the problem *instance is* chosen? Speedup is intended to measure the time taken by the best sequential algorithm divided by the time taken by the parallel algorithm, but it is going too far to allow the definition of "best" to change every time the problem instance changes. In other words, it is more realistic to assume that *the best sequential algorithm and the parallel algorithm be chosen before the particular problem instance is chosen*. In this case it is possible for the parallel algorithm to exhibit superlinear speedup for some problem instances. For example, when solving a search problem, a sequential algorithm may waste a lot of time examining a dead-end strategy. A parallel algorithm pursues many possible strategies simultaneously, and one of the processes may "luck out" and find the solution very quickly.

The second questionable assumption is that a single processor can always emulate multiple processors without a loss of efficiency. There are often architectural reasons why this assumption does not hold. For example, each CPU in a UMA multiprocessor has a certain amount of cache. A group of p processors executing a parallel algorithm has p times as much cache memory as a single processor. It is easy to construct circumstances in which *the collective cache hit rate of a group of p processors is significantly higher than the cache hit rate of a single processor* executing the best sequential algorithm or emulating the parallel algorithm. In these circumstances the p processors can execute a parallel algorithm more than p times faster than a single processor executing the best sequential algorithm.

1.6.2 Scaled speedup

Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. Amdahl's law states that the maximum speedup S achievable by a parallel computer with p processors performing the computation is

$$S \leq 1 / (f + (1 - f) / p)$$

A corollary follows immediately from Amdahl's law: a small number of sequential operations can significantly limit the speedup achievable by a parallel computer. For example, if 10 percent of the operations must be performed sequentially, then the maximum speedup achievable is 10, no matter how many processors a parallel computer has.

Amdahl's law is based upon the idea that parallel processing is used to reduce the time in which a problem of some particular size can be solved, and in some contexts this is a reasonable assumption. In many situations, however, parallel processing is used to increase the size of the problem that can be solved in a given (fixed) amount of time. For example, a design engineer checking the turbulence around an air foil may be willing to wait an hour for the computer to determine the solution. Increasing the speed of the computer allows the engineer to increase the resolution of the computed answer. Under these circumstances, Amdahl's law is not an accurate indicator of the value of parallel processing, because Amdahl's law assumes that the sequential component is fixed.

However, for many problems the proportion of the operations that are sequential decreases as the problem size increases. Every parallel program has certain overhead costs, such as creating processes, which are independent of the problem size. Other overhead costs, such as input/output and process synchronisation, may increase with the problem size, but at a slower rate than the grain size. This phenomenon is called the Amdahl effect [GoodHed77], and it explains why speedup is almost universally an increasing function of the size of the input problem (Figure 1-41).

An **algorithm is scaleable** if the level of parallelism increase at least linearly with the problem size. An **architecture is scaleable** if it continues to yield the same performance per processor, albeit used on a larger problem size, as the number of processors increases. Algorithmic and architectural scalability are important, because they allow a user to solve larger problems in the same amount of time by using a parallel computer with more processors. Data parallel algorithms are more scaleable than function parallel algorithms because the level of function parallelism is usually a constant, independent of the problem size, while the level of data parallelism is an increasing function of the problem size.

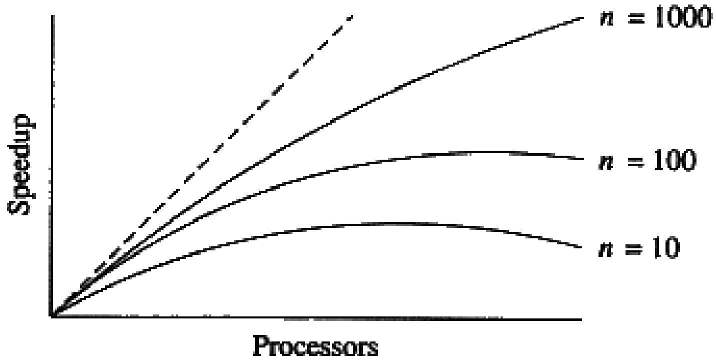


Figure 1-41 The Amdahl effect. As a general rule, speedup is an increasing function of the problem size

1.7 Data parallel approach with I/O

The prime-finding algorithms we have described in Section 4 are unrealistic, because they terminate without storing their results. Let's examine what happens when we execute a data-parallel implementation of the Sieve of Eratosthenes incorporating output on the shared-memory model of parallel computation.

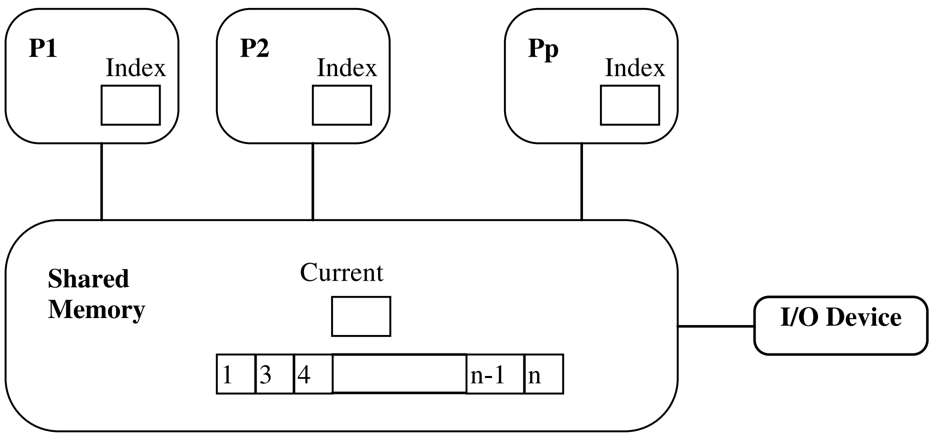


Figure 1-42 Shared-memory parallel model incorporating a sequential I/O device

The augmented shared memory model appears in Figure 1-42. Assume only one processor at a time can access the I/O device. Let $i\beta$ denote the time needed for a processor to transmit i prime numbers to that device.

Let's predict the speedup achieved by the data-parallel algorithm that outputs all primes to the I/O device. Suppose n = 1,000,000. There are 78,498 primes less than 1,000,000. To find the total execution time, we take the total computation time,

$$\left(\left\lceil \frac{\left\lceil 1{,}000{,}000/p \right\rceil}{2} \right\rceil + \left\lceil \frac{\left\lceil 1{,}000{,}000/p \right\rceil}{3} \right\rceil + \ldots + \left\lceil \frac{\left\lceil 1{,}000{,}000/p \right\rceil}{997} \right\rceil\right)\chi$$

and add to it the total I/O time, 78,498β. Figure 1-43 illustrates the two components of the parallel execution time: computation time and output time of this parallel algorithm for 1,2,...,32 processors, asuming that β = x. Because output to the I/O device must be performed sequentially, according to Amdahl's law it puts a damper on the speedup achievable through parallelization. We have seen in Section 5.2 how to calculate the maximum speedup *S* achievable by a parallel computer with *p* processors. In the current algorithm, *n* = 1,000,000, the sequential algorithm marks 2,122,048 cells and outputs 78,498 primes. Assuming these two kinds of operations take the same amount of time, the total sequential time is 2,200,546, and *f* = 78,498/2,200,S46 = .0357. An upper bound on the speedup achievable by a parallel computer with p processors is

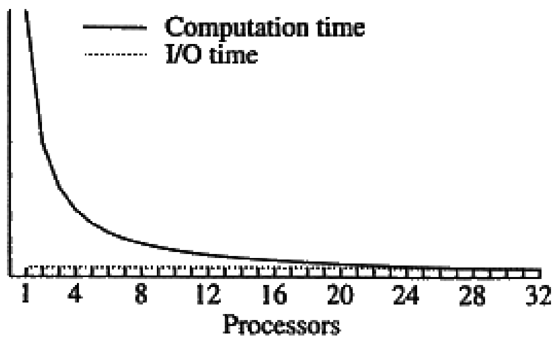


Figure 1-43 Total execution time of data-parallel output-producing Sieve of Eratosthenes algorithm as a function of its two components. This graph is for the case when n = 1,000,000 and β = x.

```
1/(.0357 + .9643/p)
```

which is shown in Figure 1-44.

Often, as the size of the problem increases, the fraction *f* of inherently sequential operations decreases, making the problem more amenable to parallelization. This phenomenon called the Amdahl effect—is true for the application we have been considering.

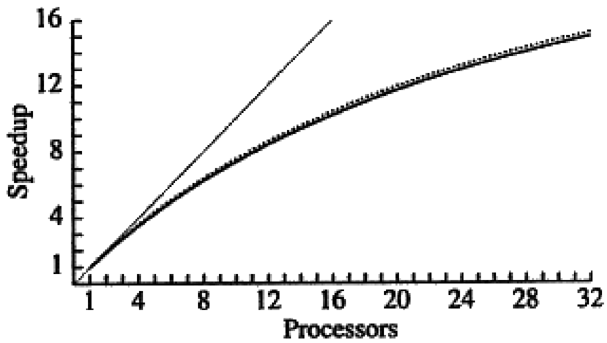


Figure 1-44 Expected speedup of data-parallel output-producing Sieve of Eratosthenes algorithm. This graph is for the case when n = 1,000,000 and β = x.