**Memory organizations**

**Interleaved Memory**

In an interleaved memory, the memory is divided into a set of banks. An interleaved memory with banks is said to be -way interleaved. One way of allocating virtual addresses to memory modules is to divide the memory space (the set of all possible addresses a processor can generate) into contiguous blocks. If there are banks, memory location would reside in bank number (ignoring remainders). In an interleaved memory, however, consecutive addresses reside in different banks: memory location is in bank number. For example, suppose there are 4 banks, each containing 256 bytes. The block-oriented scheme would assign virtual addresses, to the first bank, to the second bank, and so on. The interleaved scheme would assign addresses 0, 4, 8, to the first bank, 1, 5, 9, to the second bank, etc. (Figure 6).

However the memory space is split up among the banks, as long as requests are sent to two different banks they can be handled simultaneously. The processor can request a transfer from location on one cycle, and on the next cycle request information from location. If and are in different banks, the information will be returned on successive cycles. Note that the latency of the request, i.e. the number of cycles a processor has to wait before receiving the contents of location, is not affected. However the bandwidth is improved; if there are enough banks the memory system can potentially send information at a rate of one word per processor cycle, regardless of what the memory cycle time is.



The decision to allocate addresses as contiguous blocks or in interleaved fashion depends on how one expects information to be accessed. Programs are compiled so instructions reside in successive addresses, so there is a high probability that after a processor executes the instruction at location it will execute the instruction at (Section 2.2). Compilers can also allocate vector elements to successive addresses, so operations on entire vectors can take advantage of interleaving. For these reasons, vector processors universally have some form of interleaved memory. However, shared memory multiprocessors use the block-oriented scheme since memory referencing patterns in an MIMD system are quite different. There the goal is to connect a processor to a single memory and use as much information as possible from that memory before switching to another memory.

Systems often provide some flexibility in fetching vector elements. In some systems it is possible to load every element, for example when fetching elements of a vector v that is stored in consecutive memory cells with the memory would return, The interval between elements is known as the stride. One interesting use of this feature is in accessing matrices. If the stride is set to one more than the number of rows, a single memory request will return the diagonal elements (assuming column major layout and the columns are stored contiguously). Using a stride may cancel any benefits of interleaving if programmers are not careful. In an extreme case, setting the stride to the degree of interleaving means every item is fetched from the same bank and the time between successive elements will be the memory cycle time.



**Shared Memory**

A straightforward way to connect several processors together to build a multiprocessor is shown in Figure 7. The physical connections are quite simple. Most bus structures allow an arbitrary (but not too large) number of devices to communicate over the bus. Bus protocols were initially designed to allow a single processor and one or more disk or tape controllers to communicate with memory. If the I/O controllers are replaced by processors, one has a small single-bus multiprocessor.

The problem with this design is that processors must contend for access to the bus. If a processor P is fetching an instruction, all other processors P must wait until the bus is free. If there are only two processors they can perform close to their maximum rate since the bus can alternate between them: as one processor is decoding and executing an instruction, the other can be using the bus to fetch its next instruction. However, when a third processor is added performance begins to degrade. Usually by the time 10 processors are connected to the bus the performance curve has flattened out so that adding an 11th processor will not increase performance at all. The bottom line is the fact that the memory and bus have a fixed bandwidth, determined by a combination of the cycle time of the memory and the bus protocol, and in a single-bus multiprocessor this bandwidth is divided among several processors. If the processor cycle time is very slow compared to the memory cycle, a fairly large number of processors can be accommodated by this plan, but in fact processor cycles are usually much faster than memory cycles so this scheme is not widely used.

A slight modification to this design will improve performance, but it cannot indefinitely postpone the flattening of the performance curve. If each processor has its own local cache, there is a high probability () that the instruction or data it wants is in the local cache. A reasonable cache hit rate will greatly reduce the number of accesses a processor makes and thus improve overall efficiency. The ``knee'' of the performance curve, which identifies a point where it is still cost-effective to add processors, can now be around 20 processors, and the curve will not flatten out until around 30 processors.

Giving each processor its own cache introduces a difficulty known as the cache coherency problem. In its simplest form, the problem may be exemplified by the following scenario. Suppose two processors use data item A, so A ends up in the cache of both processors. Next suppose processor 1 performs a calculation that changes A. When it is done, the new value of A is written out to main memory. Processor 2 at a later time needs to fetch A. However, since A was already in its cache, it will use the cached value and not the newly updated value calculated by processor 1. Maintaining a consistent version of shared data requires providing new versions of the cached data to each processor whenever one of the processors updates its copy.

The multiprocessors produced by Sequent, Inc. are classic examples of machines of this type. Their first machine, the Balance 8000, was intended to compete with the DEC VAX 780, a popular minicomputer at that time. A 2-processor configuration gave slightly less performance than the VAX, but the next larger configuration, with four processors, was faster. The operating system was a modified version of UNIX. There was a single global task queue, and each processor could fetch a task from the queue, execute it until it blocks or times out, and return it to the queue. Thus the system implemented a form of job level parallelism. Sequent also provided a library of procedures that allowed users to write parallel programs, and the machine became a popular test bed for parallel languages and algorithms. The current machines, in the Symmetry series, are widely used for on-line transaction processing.



Programming a shared memory machine is fairly straightforward. Programming constructs such as semaphores, fork-join, and monitors, which were developed for communication and synchronization of parallel processes in operating systems and other concurrent programming applications, have been adapted for parallel processing. The implementation of the basic synchronization primitives from which these constructs are built is more complex in a parallel system, but this complexity is hidden from users. For example, the bus in the Sequent Symmetry has provisions for implementing a pool of semaphores so that processes are guaranteed to gain exclusive access to shared structures.

Another way of building a shared memory multiprocessor is shown in Figure 8. In these designs, the bus has been replaced by a switch that routes requests from a processor to one of several different memory modules. Even though there are several physical memories, there is one large virtual address space. The advantage of this organization is based on the fact the switch can handle multiple requests in parallel. Each processor can be paired up with a memory, and each can then run at full speed as it accesses the memory it is currently connected to. Contention still occurs, though. If two processors make requests of the same memory module only one will be given access and the other will be blocked. Several machines with this design will be discussed in the survey of MIMD machines following the section on interconnection topology, which introduces concepts that will explain various switch designs.

###

### Distributed Memory

### In a distributed memory system the memory is associated with individual processors and a processor is only able to address its own memory. Some authors refer to this type of system as a multicomputer, reflecting the fact that the building blocks in the system are themselves small computer systems complete with processor and memory.

### There are several benefits of this organization. First, there is no bus or switch contention. Each processor can utilize the full bandwidth to its own local memory without interference from other processors. Second, the lack of a common bus means there is no inherent limit to the number of processors; the size of the system is now constrained only by the network used to connect processors to each other. Third, there are no cache coherency problems. Each processor is in charge of its own data, and it does not have to worry about putting copies of it in its own local cache and having another processor reference the original.

### The major drawback in the distributed memory design is that interprocessor communication is more difficult. If a processor requires data from another processor's memory, it must exchange messages with the other processor. This introduces two sources of overhead: it takes time to construct and send a message from one processor to another, and a receiving processor must be interrupted in order to deal with messages from other processors.

### Programming on a distributed memory machine is a matter of organizing a program as a set of independent tasks that communicate with each other via messages. In addition, programmers must be aware of where data is stored, which introduces the concept of locality in parallel algorithm design. An algorithm that allows data to be partitioned into discrete units and then runs with minimal communication between units will be more efficient than an algorithm that requires random access to global structures.

### Semaphores, monitors, and other concurrent programming techniques are not directly applicable on distributed memory machines, but they can be implemented by a layered software approach. User code can invoke a semaphore, for example, which is itself implemented by passing a message to the node that ``owns'' the semaphore. This approach is not very efficient, however, and it has the drawback of nonuniform memory access, i.e. the latency of a memory request, in this case reading the value of a semaphore, is proportional to the distance between the processor making the request and the memory where the value is stored.

### Which programming style is easier - shared memory with semaphores, etc. or distributed memory with message passing - is often a matter of personal preference. The message passing style fits very well with the object oriented programming methodology, and if a program is already organized in terms of objects it may be quite easy to adapt it for a distributed memory system. When faced with a decision of whether to implement a program in shared memory or distributed memory the outcome is usually based on the amount of information that must be shared by parallel tasks. Whatever information is shared among tasks must be copied from one node to another via messages in a distributed memory system, and this overhead may reduce efficiency to the point where a shared memory system is preferred.

### http://www.phy.ornl.gov/csep/gif_figures/caf9.gif

A PMS diagram of a simple distributed memory parallel processor is shown in Figure 9. On the left is the diagram of a single node, often called a processing element, or PE. The organization of a PE explains how messages are passed from one PE to another. As far as any one processor is concerned, the other processors are simply I/O devices. To send a message to another PE, a processor copies information into a data block in its local memory and then tells its local controller to transfer the information to an external device, much the same way a disk controller in a microcomputer would write a block on a disk drive. In this case, however, the block of data is transferred over the interconnection network to an I/O controller in the receiving node. That controller finds room for the incoming message in its local memory and then notifies the processor that a message has arrived.