Part IV.

Chapter 15 - Introduction to MIMD Architectures

Thread and process-level parallel architectures are typically realised by MIMD (Multiple Instruction Multiple Data) computers. This class of parallel computers is the most general one since it permits autonomous operations on a set of data by a set of processors without any architectural restrictions. Instruction level data-parallel architectures should satisfy several constraints in order to build massively parallel systems. For example processors in array processors, systolic architectures and cellular automata should work synchronously controlled by a common clock. Generally the processors are very simple in these systems and in many cases they realise only a special function (systolic arrays, neural networks, associative processors, etc.). Although in recent SIMD architectures the complexity and generality of the applied processors have been increased, these modifications have resulted in the introduction of process-level parallelism and MIMD features into the last generation of data-parallel computers (for example CM-5), too.

MIMD architectures became popular when progress in integrated circuit technology made it possible to produce microprocessors which were relatively easy and economical to connect into a multiple processor system. In the early eighties small systems, incorporating only tens of processors were typical. The appearance of Transputer in the mid-eighties caused a great breakthrough in the spread of MIMD parallel computers and even more resulted in the general acceptance of parallel processing as the technology of future computers. By the end of the eighties mid-scale MIMD computers containing several hundreds of processors become generally available. The current generation of MIMD computers aim at the range of massively parallel systems containing over 1000 processors. These systems are often called scalable parallel computers.

15.1 Architectural concepts

The MIMD architecture class represents a natural generalisation of the uniprocessor von Neumann machine which in its simplest form consists of a single processor connected to a single memory module. If the goal is to extend this architecture to contain multiple processors and memory modules basically two alternative choices are available:

a. The first possible approach is to replicate the processor/memory pairs and to connect them via an interconnection network. The processor/memory pair is called

processing element (PE) and they work more or less independently of each other. Whenever interaction is necessary among the PEs they send messages to each other. None of the PEs can ever access directly the memory module of another PE. This class of MIMD machines are called the **Distributed Memory MIMD Architectures** or **Message-Passing MIMD Architectures**. The structure of this kind of parallel machines is depicted in Figure 1.

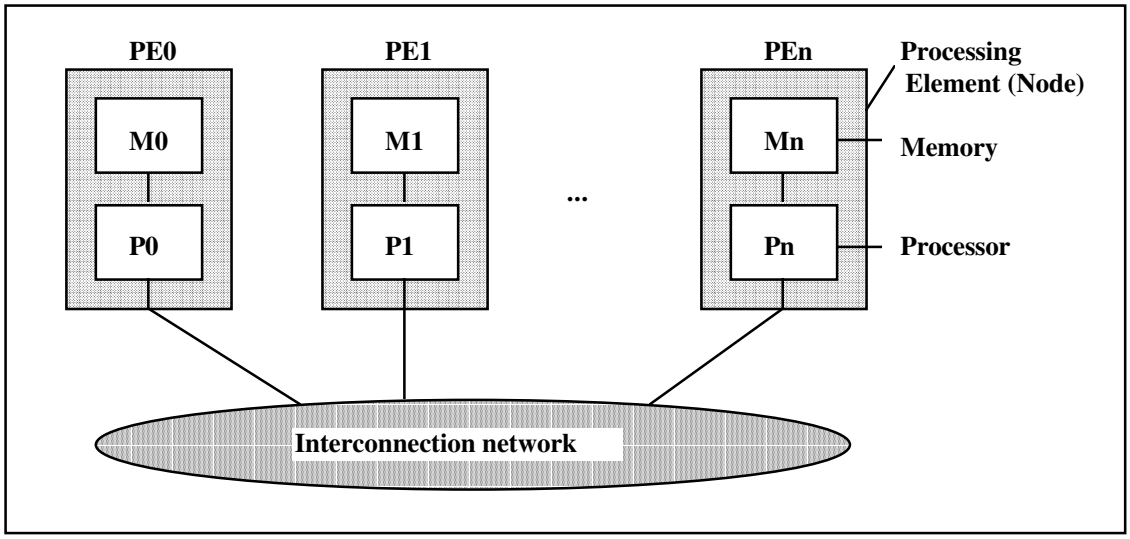

Figure 1. Structure of Distributed Memory MIMD Architectures

b. The second alternative approach is to create a set of processors and memory modules. Any processor can directly access any memory modules via an interconnection network as it is shown in Figure 2. The set of memory modules defines a global address space which is shared among the processors. The name of this kind of parallel machines is **Shared Memory MIMD Architectures** and this arrangement of processors and memory is called the **dance-hall** shared memory system.

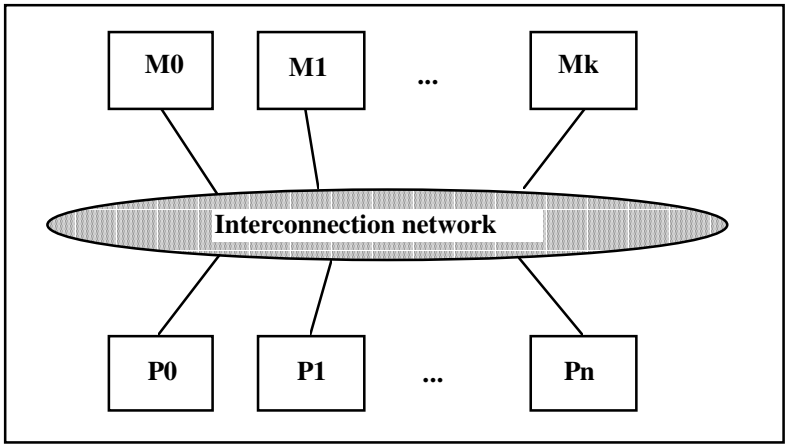

Figure 2. Structure of Shared Memory MIMD Architectures

Distributed Memory MIMD Architectures are often simply called **multicomputers** while Shared Memory MIMD Architectures are shortly referred as **multiprocessors**. In

both architecture types one of the main design considerations is how to construct the interconnection network in order to reduce message traffic and memory latency. A network can be represented by a **communication graph** in which vertices correspond to the switching elements of the parallel computer and edges represent communication links. The topology of the communication graph is an important property which significantly influents latency in parallel computers. According to their topology interconnection networks can be classified as **static** and **dynamic networks**. In static networks the connection of switching units is fixed and typically realised as direct or **point-to-point connections**. These networks are called **direct networks**, too. In dynamic networks communication links can be reconfigured by setting the active switching units of the system. Multicomputers are typically based on static networks, while dynamic networks are mainly employed in multiprocessors. It should be pointed out here that the role of interconnection networks is different in distributed and shared memory systems. In the former one the network should transfer complete messages which can be of any length and hence special attention should be paid to support message passing protocols. In shared memory systems short but frequent memory accesses are the typical way of using the network. Under these circumstances special care is needed to avoid **contention** and **hot spot** problems in the network.

There are some advantages and drawbacks of both architecture types. The advantages of the distributed memory systems are:

1. Since processors work on their attached local memory module most of the time, the contention problem is not so severe as in the shared memory systems. As a result distributed memory multicomputers are highly scalable and good architectural candidates of building massively parallel computers.

2. Processes cannot communicate through shared data structures and hence sophisticated synchronisation techniques like monitors are not needed. Message passing solves not only communication but synchronisation as well.

Most of the problems of distributed memory systems come from the programming side:

1. In order to achieve high performance in multicomputers special attention should be paid to load balancing. Although recently large research effort has been devoted to provide automatic mapping and load balancing, it is still the responsibility of the user to partition the code and data among the PEs in many systems.

2. Message-passing based communication and synchronisation can lead to deadlock situations. On the architecture level it is the task of the communication protocol designer to avoid deadlocks derived from incorrect routing schemes. However, to avoid deadlocks

of message based synchronisation at the software level is still the responsibility of the user.

3. Though there is no architectural bottleneck in multicomputers, message-passing requires the physical copy of data structures among processes. Intensive data copying can result in significant performance degradation. This was the case in particular for the first generation of multicomputers where the applied store-and-forward switching technique consumed both processor time and memory space. The problem is radically reduced by the second generation of multicomputers where introduction of the wormhole routing and employment of special purpose communication processors resulted in an improvement of three orders of magnitude in communication latency.

Advantages of shared memory systems appear mainly in the field of programming these systems:

1. There is no need to partition either the code or the data, therefore programming techniques applied for uniprocessors can easily be adapted in the multiprocessor environment. Neither new programming languages nor sophisticated compilers are needed to exploit shared memory systems.

2. There is no need to physically move data when two or more processes communicate. The consumer process can access the data on the same place where the producer composed it. As a result communication among processes is very efficient.

Unfortunately there are several drawbacks in the case of shared memory systems, too:

1. Although programming shared memory systems is generally easier than programming multicomputers the synchronised access of shared data structures requires special synchronising constructs like semaphores, conditional critical regions, monitors, etc. The use of these constructs results in nondeterministic program behaviour which can lead to programming errors that are difficult to discover. Usually message passing synchronisation is simpler to understand and apply.

2. The main disadvantage of shared memory systems is the lack of scalability due to the contention problem. When several processors want to access the same memory module they should compete for the right to access the memory. Meanwhile the winner can access the memory, the losers should wait for the access right. The larger the number of processors, the probability of memory contention is higher. Beyond a certain number of processors the probability is so high in a shared memory computer that adding a new processor to the system will not increase the performance.

There are several ways to overcome the problem of low scalability of shared memory systems:

1. The use of high through-put, low-latency interconnection network among the processors and memory modules can significantly improve scalability.

2. In order to reduce the memory contention problem shared memory systems are extended with special, small size local memories called as **cache memories**. Whenever a memory reference is given by a processor, first the attached cache memory is checked if the required data is stored in the cache. If yes, the memory reference can be performed without using the interconnection network and as a result the memory contention is reduced. If the required data is not in the cache memory, the page containing the data is transferred to the cache memory. The main assumption here is that shared-memory programs generally provide good locality of reference. For example, during the execution of a procedure in many cases it is enough to access only the local data of the procedure which are all contained in the cache of the performing processor. Unfortunately, many times this is not the case, which reduces the ideal performance of cache extended shared memory systems. Furthermore a new problem, called the **cache coherence problem** appears, which further limits the performance of cache based systems. The problems and solutions of cache coherence will be discussed in detail in Chapter 18.

3. The logically shared memory can be physically implemented as a collection of local memories. This new architecture type is called **Virtual Shared Memory** or **Distributed Shared Memory Architecture**. From the point of view of physical construction a distributed shared memory machine resembles very much to a distributed memory system. The main difference between the two architecture types comes from the organisation of the address space of the memory. In the distributed shared memory systems the local memories are components of a global address space and any processor can access the local memory of any other processors. In distributed memory systems the local memories have separate address spaces and direct access of the local memory of a remote processor is prohibited.

Distributed shared memory systems can be divided into three classes based on the access mechanism of the local memories:

1. Non-Uniform-Memory-Access (**NUMA)** machines
2. Cache-Coherent Non-Uniform-Memory-Architecture (**CC-NUMA**) machines
3. Cache-Only Memory Architecture (**COMA**) machines

The general structure of NUMA machines is shown in Figure 3. A typical example of this architecture class is the Cray T3D machine. In NUMA machines the shared memory is divided into as many blocks as many processors are in the system and each memory block is attached to a processor as a local memory with direct bus connection. As a result whenever a processor addresses the part of the shared memory that is connected as local memory, the access of that block is much faster than the access of the

remote ones. This non-uniform access mechanism requires careful program and data distribution among the memory blocks in order to really exploit the potential high performance of these machines. Consequently NUMA architectures have similar drawbacks to the distributed memory systems. The main difference between them appears in the programming style: meanwhile distributed memory systems are programmed based on the message passing paradigm, programming of the NUMA machines still relies on the more conventional shared memory approach. However, in recent NUMA machines like in the Cray T3D, message passing library is available, too and hence, the difference between multicomputers and NUMA machines became close to negligible.

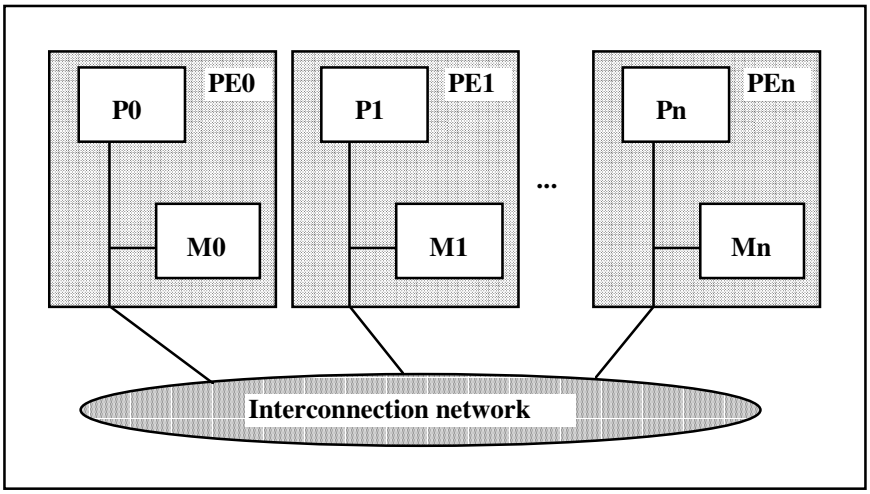

Figure 3. Structure of NUMA Architectures

The other two classes of distributed shared memory machines employ **coherent caches** in order to avoid the problems of NUMA machines. The single address space and coherent caches together significantly ease the problem of data partitioning and dynamic load balancing, providing better support for multiprogramming and parallelising compilers. They differ in the extent of applying coherent caches. In COMA machines every memory block works as a cache memory. Based on the applied cache coherence scheme data dynamically and continuously migrate to the local caches of those processors where the data are most needed. Typical examples are the KSR-1 and the DDM machines. The general structure of COMA machines is depicted in Figure 4.

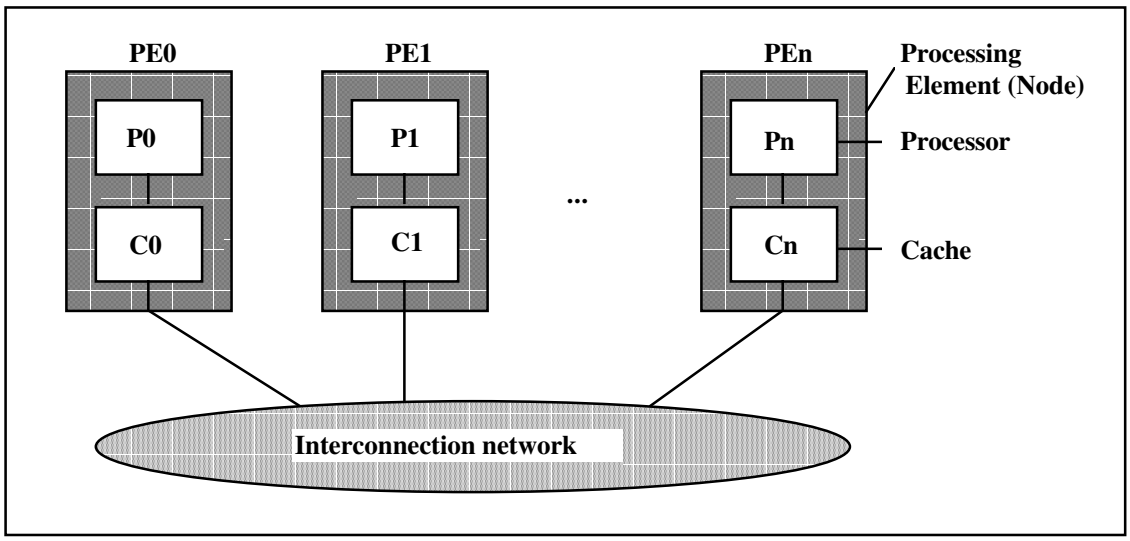

Figure 4. Structure of COMA Architectures

CC-NUMA machines represent a compromise between the NUMA and COMA machines. Like in the NUMA machines the shared memory is constructed as a set of local memory blocks. However, in order to reduce the traffic of the interconnection network each processor node is supplied with a large cache memory block. Though the initial data distribution is static like in the NUMA machines, dynamic load balancing is achieved by the cache coherence protocols like in the COMA machines. Most of the current massively parallel distributed shared memory machines are built on the concept of CC-NUMA architectures. Examples are Convex SPP1000, Stanford DASH and MIT Alewife. The general structure of CC-NUMA machines is shown in Figure 5.

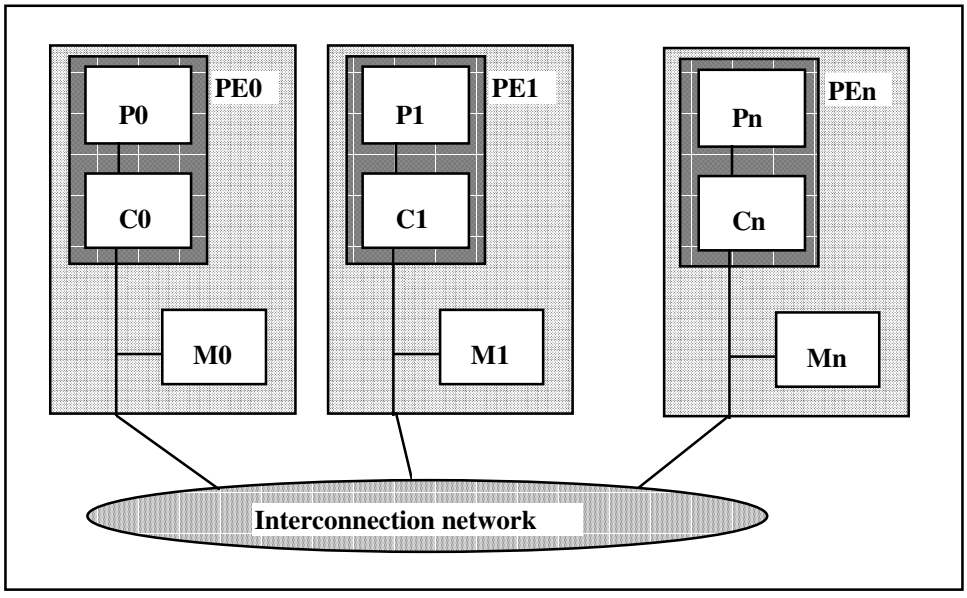

Figure 5. Structure of CC-NUMA Architectures

Process-level architectures have been realised either by multiprocessors or by multicomputers. Interestingly in case of thread-level architectures only shared memory

systems have been built or proposed. The classification of MIMD computers are depicted in Figure 6. Details of the multithreaded architectures, distributed memory and shared memory systems are given in detail in the forthcoming chapters.

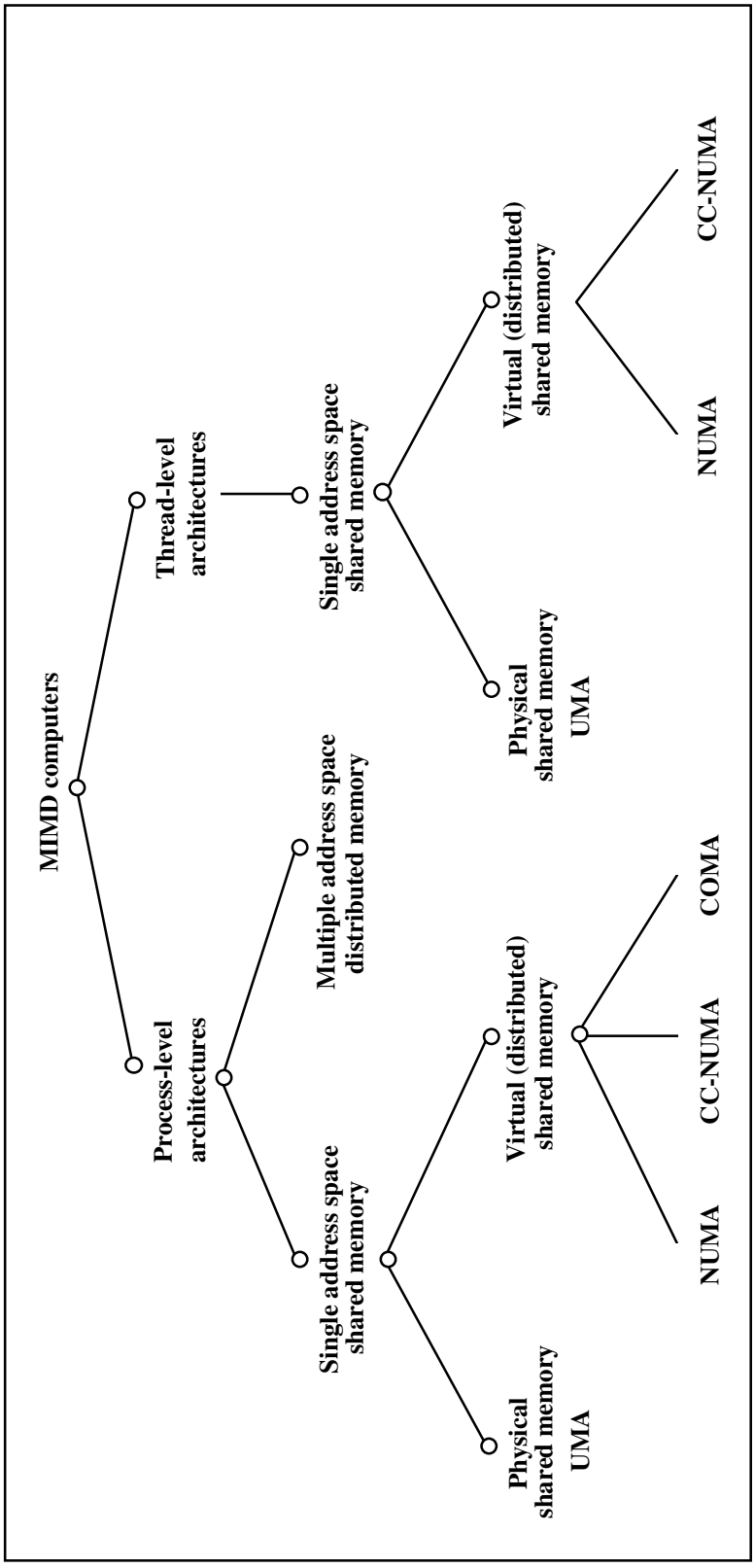

Figure 6. Classification of MIMD computers

15.2 Problems of scalable computers

There are two fundamental problems to be solved in any scalable computer system (Arvind and Iannucci, 1987):

1. tolerate and hide latency of remote loads

2. tolerate and hide idling due to synchronisation among parallel processes.

Remote loads are unavoidable in scalable parallel systems which use some form of distributed memory. Accessing a local memory usually requires only one clock cycle while access to a remote memory cell can take two orders of magnitude longer time. If a processor issuing such a remote load operation should wait for the completeness of the operation without doing any useful work, the remote load would significantly slow down the computation. Since the rate of load instructions is high in usual programs, the latency problem would eliminate all the potential benefits of parallel activities. A typical case is shown if Figure 7. where P0 has to load two values A and B from two remote memory block M1 and Mn in order to evaluate the expression A+B. The pointers to A and B are rA and rB stored in the local memory of P0. Access of A and B are realised by the "rload rA" and "rload rB" instructions that should travel through the interconnection network in order to fetch A and B.

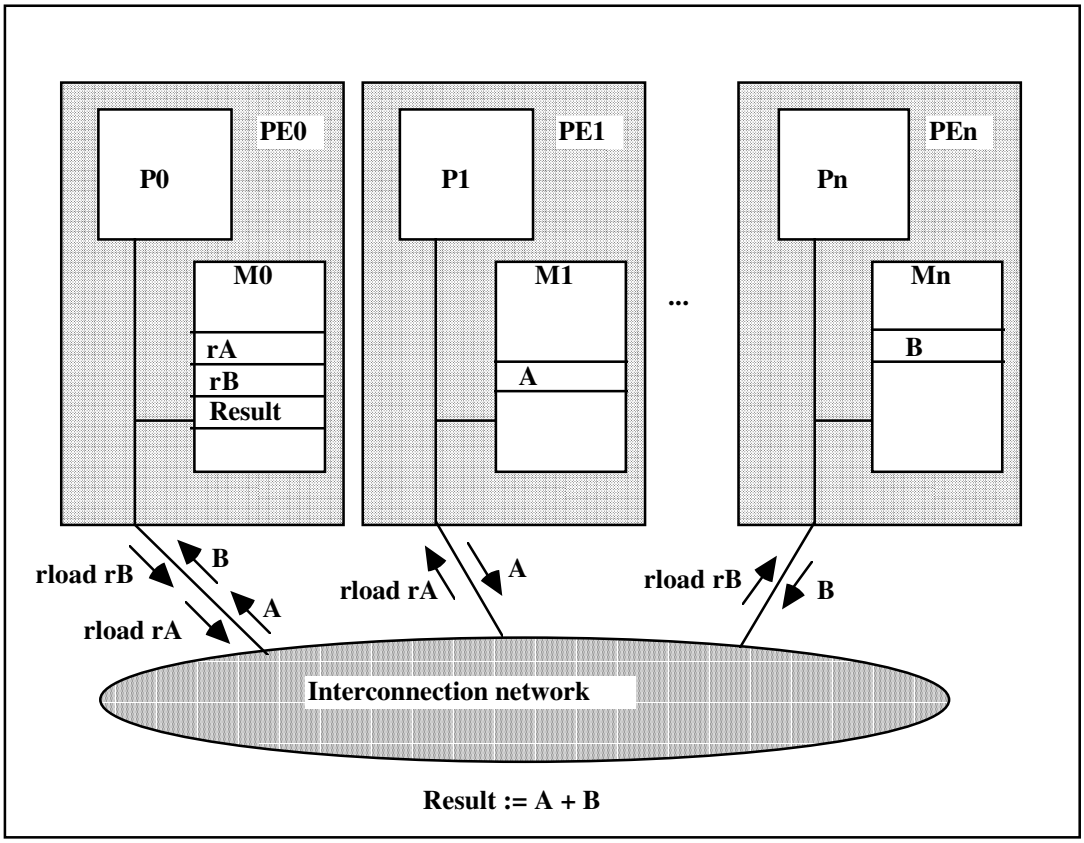

Figure 7. The remote load problem

The situation is even worse if the values of rA and rB are currently not available in M1 and Mn since they are subject of to be produced by certain other processes to run later on. In this case where idling occurs due to synchronisation among parallel processes, the original process on P0 should wait unpredictable time resulting in **unpredictable latency**.

In order to solve the above-mentioned problems several possible hardware/software solutions were proposed and applied in various parallel computers:

1. application of cache memory

2. prefetching

3. introduction of threads and fast context switching mechanism among threads.

Application of **cache memory** greatly reduces the time spent on remote load operations if most of the load operations can be performed on the local cache. Suppose that A is placed in the same cache block as C and D that are objects in the expression following the one that contains A:

```
Result := A + B;
Result2 := C - D;
```

Under such circumstances caching A will bring C and D to the cache memory of P0 and hence, the remote load of C and D is replaced by local cache operations that cause significant acceleration in the program execution.

The **prefetching** technique, too relies on a similar principle. The main idea is to bring data to the local memory or cache before it is actually needed. A prefetch operation is an explicit nonblocking request to fetch data before the actual memory operation is issued. The remote load operation applied in the prefetch does not slow down the computation since the data to be prefetched will be used only later and hopefully, by the time the requiring process needs the data, its value has been brought closer to the requesting processor, hiding the latency of the usual blocking read.

Notice that these solutions can not solve the problem of idling due to synchronisation. Even for remote loads cache memory can not reduce latency in every case. At cache miss the remote load operation is still needed and moreover **cache coherence** should be maintained in parallel systems. Obviously, maintenance algorithms for cache coherence reduce the speed of cache based parallel computers.

The third approach - introducing **threads and fast context switching** mechanisms - offers a good solution for both the remote load latency problem and for the synchronisation latency problem. This approach led to the construction of multithreaded

computers that are the subject of Chapter 16. A combined application of the three approaches can promise an efficient solution for both latency problems.

15.3 Main design issues of scalable MIMD computers

The main design issues in scalable parallel computers are as follows:

1. Processor design
2. Interconnection network design
3. Memory system design
4. I/O system design

The current generation of commodity processors contain several built-in parallel architecture features like pipelining, parallel instruction issue logic, etc. as it was shown in Part II. They also directly support the built of small- and mid-size multiple processor systems by providing atomic storage access, prefetching, cache coherency, message passing, etc. However, they can not tolerate remote memory load and idling due to synchronisation which are the fundamental problems of scalable parallel systems. To solve these problems a new approach is needed in processor design. Multithreaded architectures described in detail in Chapter 16 offer a promising solution in the very near future.

Interconnection network design was a key problem in the data-parallel architectures since they aimed at massively parallel systems, too. Accordingly, the basic interconnections of parallel computers have been described in Part III. In the current part those design issues will be reconsidered that are relevant for the case when commodity microprocessors are to be applied in the network. Particularly, Chapter 17 is devoted to these questions since the central design issue in distributed memory multicomputers is the selection of the interconnection network and the hardware support of message passing through the network.

Memory design is the crucial topic in shared memory multiprocessors. In these parallel systems the maintenance of a logically shared memory plays a central role. Early multiprocessors applied physically shared memory which become a bottleneck in scalable parallel computers. Recent generation of multiprocessors employs a distributed shared memory supported by distributed cache system. The maintenance of cache coherency is a nontrivial problem which requires careful hardware/software design. Solutions of the cache coherence problem and other innovative features of contemporary multiprocessors are described in the last chapter of this part.

In scalable parallel computers one of the main problems is the handling of I/O devices in an efficient way. The problem seems to be particularly serious when large data volumes should be moved among I/O devices and remote processors. The main question is how to avoid the disturbance of the work of internal computational processors. The problem of I/O system design appears in every class of MIMD systems and hence it will be discussed throughout the whole part when it is relevant.