IV. Parallel Operating Systems

João Garcia, Paulo Ferreira, and Paulo Guedes

IST/INESC, Lisbon, Portugal

Summary. Parallel operating systems are the interface between parallel computers (or computer systems) and the applications (parallel or not) that are executed on them. They translate the hardware's capabilities into concepts usable by programming languages.

Great diversity marked the beginning of parallel architectures and their operating systems. This diversity has since been reduced to a small set of dominating configurations: symmetric multiprocessors running commodity applications and operating systems (UNIX and Windows NT) and multicomputers running custom kernels and parallel applications. Additionally, there is some (mostly experimental) work done towards the exploitation of the shared memory paradigm on top of networks of workstations or personal computers.

In this chapter, we discuss the operating system components that are essential to support parallel systems and the central concepts surrounding their operation: scheduling, synchronization, multi-threading, inter-process communication, memory management and fault tolerance.

Currently, SMP computers are the most widely used multiprocessors. Users find it a very interesting model to have a computer, which, although it derives its processing power from a set of processors, does not require any changes to applications and only minor changes to the operating system. Furthermore, the most popular parallel programming languages have been ported to SMP architectures enabling also the execution of demanding parallel applications on these machines.

However, users who want to exploit parallel processing to the fullest use those same parallel programming languages on top of NORMA computers. These multicomputers with fast interconnects are the ideal hardware support for message

passing parallel applications. The surviving commercial models with NORMA architectures are very expensive machines, which one will find running calculus intensive applications, such as weather forecasting or fluid dynamics modelling.

We also discuss some of the experiments that have been made both in hardware (DASH, Alewife) and in software systems (TreadMarks, Shasta) to deal with the scalability issues of maintaining consistency in shared-memory systems and to prove their applicability on a large-scale.

1. Introduction

Parallel operating systems are primarily concerned with managing the resources of parallel machines. This task faces many challenges: application programmers demand all the performance possible, many hardware configurations exist and change very rapidly, yet the operating system must increasingly be compatible with the mainstream versions used in personal computers and workstations due both to user pressure and to the limited resources available for developing new versions of these system.

In this chapter, we describe the major trends in parallel operating systems. Since the architecture of a parallel operating system is closely influenced by the hardware architecture of the machines it runs on, we have divided our presentation in three major groups: operating systems for small scale symmetric multiprocessors (SMP), operating system support for large scale distributed memory machines and scalable distributed shared memory machines.

The first group includes the current versions of Unix and Windows NT where a single operating system centrally manages all the resources. The second group comprehends both large scale machines connected by special purpose interconnections and networks of personal computers or workstations, where the operating system of each node locally manages its resources and collaborates with its peers via message passing to globally manage the machine. The third group is composed of non-uniform memory access (NUMA) machines where each node's operating system manages the local resources and interacts at a very fine grain level with its peers to manage and share global resources.

We address the major architectural issues in these operating systems, with a greater emphasis on the basic mechanisms that constitute their core: process management, file system, memory management and communications. For each component of the operating system, the presentation covers its architecture, describes some of the interfaces included in these systems to provide programmers with a greater level of flexibility and power in exploiting the hardware, and addresses the implementation efforts to modify the operating system in order to explore the parallelism of the hardware. These topics include support for multi-threading, internal parallelism of the operating system components, distributed process management, parallel file systems, inter-process communication, low-level resource sharing and fault isolation. These features are illustrated with examples from both commercial operating systems and research proposals from industry and academia.

2. Classification of Parallel Computer Systems

2.1 Parallel Computer Architectures

Operating systems were created to present users and programmers with a view of computers that allows the use of computers abstracting away from the

details of the machine's hardware and the way it operates. *Parallel operating systems* in particular enable user interaction with computers with parallel architectures. The physical architecture of a computer system is therefore an important starting point for understanding the operating system that controls it. There are two famous classifications of parallel computer architectures: Flynn's [Fly72] and Johnson's [Joh88].

2.1.1 Flynn's classification of parallel architectures. Flynn divides computer architectures along two axes according to the number of data sources and the number of instruction sources that a computer can process simultaneously. This leads to four categories of computer architectures:

SISD - Single Instruction Single Data. This is the most widespread architecture where a single processor executes a sequence of instructions that operate on a single stream of data. Examples of this architecture are the majority of computers in use nowadays such as personal computers (with some exceptions such as Intel Dual Pentium machines), workstations and low-end servers. These computers typically run either a version of the Microsoft Windows operating system or one of the many variants of the UNIX operating system (Sun Microsystems' Solaris, IBM's AIX, Linux...) although these operating systems include support for multiprocessor architectures and are therefore not strictly operating systems for SISD machines.

MISD - Multiple Instruction Single Data. No existing computer corresponds to the MISD model, which was included in this description mostly for the sake of completeness. However, we could envisage special purpose machines that could use this model. For example, a "cracker computer" with a set of processors where all are fed the same stream of ciphered data which each tries to crack using a different algorithm.

SIMD - Single Instruction Multiple Data. The SIMD architecture is used mostly for supercomputer vector processing and calculus. The programming model of SIMD computers is based on distributing sets of homogeneous data among an array of processors. Each processor executes the same operation on a fraction of a data element. SIMD machines can be further divided into pipelined and parallel SIMD. Pipelined SIMD machines execute the same instruction for an array of data elements by fetching the instruction from memory only once and then executing it in an overloaded way on a pipelined high performance processor (e.g. Cray 1). In this case, each fraction of a data element (vector) is in a different stage of the pipeline. In contrast to pipelined SIMD, parallel SIMD machines are made from dedicated processing elements connected by high-speed links. The processing elements are special purpose processors with narrow data paths (some as narrow as one bit) and which execute a restricted set of operations. Although there is a design effort to keep processing elements as cheap as possible, SIMD machines are quite rare due to their high cost, limited applicability and to the fact that they are virtually impossible to upgrade, thereby representing a "once in a lifetime" investment for many companies or institutions that buy one.

An example of a SIMD machine is the Connection Machine (e.g. CM-2) [TMC90a], built by the Thinking Machines Corporation. It has up to 64k processing elements where each of them operates on a single bit and is only able to perform very simple logical and mathematical operations. For each set of 32 processing cells there are four memory chips, a numerical processor for floating point operations and a router connected to the cell interconnection. The cells are arranged in a 12-dimensional hyper-cube topology. The concept of having a vast amount of very simple processing units served three purposes: to eliminate the classical memory access bottleneck by having processors distributed all over the memory, to experiment with the scalability limits of parallelism and to approximate the brain model of computation.

The MasPar MP-2 [Bla90] by the MasPar Computer Corporation is another example of a SIMD machine. This computer is based on a 2 dimensional lattice of up to 16k processing elements driven by a VAX computer that serves as a front end to the processing elements. Each processing element is connected to its eight neighbours on the 2D mesh to form a torus. The user processes are executed on the front-end machine that passes the parallel portions of the code to another processor called the Data Parallel Unit (DPU). The DPU is responsible for executing operations on singular data and for distributing the operations on parallel data among the processing elements that constitute the lattice.

MIMD - Multiple Instruction Multiple Data. These machines constitute the core of the parallel computers in use today. The MIMD design is a more general one where machines are composed of a set of processors that execute different programs which access their corresponding datasets[1]. These architectures have been a success because they are typically cheaper than special purpose SIMD machines since they can be built with off-the-shelf components. To further detail the presentation of MIMD machines it is now useful to introduce Johnson's classification of computer architectures.

2.1.2 Johnson's classification of parallel architectures. *Johnson's classification* [Joh88] is oriented towards the different memory access methods. This is a much more practical approach since, as we saw above, all but the MIMD class of Flynn are either virtually extinct or never existed. We take the opportunity of presenting Johnson's categories of parallel architectures to give some examples of MIMD machines in each of those categories. Johnson divides computer architectures into:

UMA – Uniform Memory Access. UMA machines guarantee that all processors use the same mechanism to access all memory positions and that those accesses are performed with similar performance. This is achieved with a shared memory architecture where all processors access a central memory unit. In a UMA machine, processors are the only parallel element (apart from caches which we discuss below). The remaining architecture components

[1] although not necessary in mutual exclusion.

(main memory, file system, networking...) are shared among the processors. Each processor has a local cache where it stores recently accessed memory positions. Most major hardware vendors sell UMA machines in the form of symmetric multiprocessors with write-through caches. This means that writes to cache are immediately written to main memory. In the alternative policy of write-back caching, writes to the cache are only written to main memory when the cache line containing the write is evicted from the processor's cache.

In order to keep these processor caches coherent and thereby give each processor a consistent view of the main memory a hardware coherence protocol is implemented. There are two techniques for maintaining cache coherence: directory based and bus snooping protocols. Bus snooping is possible only when the machine's processors are connected by a common bus. In this case, every processor can see all memory accesses requested on the common bus and update its own cache accordingly. When snooping is not possible, a directory-based protocol is used. These protocols require that processor keeps a map of how memory locations are distributed among all processors in order to be able find it among the machine's processors when requested. Typically, bus snooping is used in shared memory SMPs and directory-based protocols are used in distributed memory architectures such as the SGI Origin 2000.

The Alpha Server family [Her97], developed by the Digital Equipment Corporation (now owned by Compaq), together with Digital UNIX has been one of the most commercially successful server and operating system solutions. Alpha Servers are symmetric multiprocessor based on Alpha processors, up to 4 of them, connected by a snoopy bus to a motherboard with up to 4 GByte of memory. The Alpha Server has separate address and data buses and uses a cache write-back protocol. When a processor writes to its cache, the operation is not immediately passed on to main memory but only when that data element has to be used by another memory block. Therefore, processors have to snoop the bus in order to reply to other processors that might request data whose more recent version is in their cache and not in main memory.

A more recent example of a UMA multiprocessor is the Enterprise 10000 Server [Cha98], which is Sun Microsystems' latest high-end server. It is configured with 16 interconnected system boards. Each board can have up to 4 UltraSPARC 336MHz processors, up to 4GB of memory and 2 I/O connections. All boards are connected to four address buses and to a 16x16 connection Gigaplane-XB interconnection matrix. This architecture, called Starfire, is a mixture between a point- to-point processor/memory topology and a snoopy bus. The system board's memory addresses are interleaved among the four address buses, which are used to send packets with memory requests from the processors to the memory. The requests are snooped by all processors connected to the bus, which allows them to implement a snooping coherency protocol. Since all processors snoop all requests and update their caches, the replies from the memory with the data requested by the processors do not have to be seen and are sent on a dedicated high-throughput

Gigaplane connection between the memory bank and the processor thereby optimising memory throughput. Hence, the architecture's bottleneck is at the address bus, which can't handle as many requests as the memory and the Gigaplane interconnection. The Enterprise 10000 runs Sun's Solaris operating system.

NUMA – Non-Uniform Memory Access. NUMA computers are machines where each processor has its own memory. However, the processor interconnection allows processors to access another processor's local memory. Naturally, an access to local memory has a much better performance than a read or write of a remote memory on another processor. The NUMA category includes replicated memory clusters, massively parallel processors, cache coherent NUMA (CC-NUMA such as the SGI Origin 2000) and cache only memory architecture. Two recent examples of NUMA machines are the SGI Origin 2000 and Cray Research's T3E. Although NUMA machines give programmers a global memory view of the processors aggregate memory, performance conscious programmers still tend to try to locate their data on the local memory portion and avoid remote memory accesses. This is one of the arguments used in favour of MIMD shared memory architectures.

The SGI Origin 2000 [LL94] is a cache coherent NUMA multiprocessor designed to scale up to 512 nodes. Each node has one or two R10000 195MHz processors with up to 4GB of coherent memory and a memory coherence controller called the hub. The nodes are connected via a SGI SPIDER router chip which routes traffic between six Craylink network connections. These connections are used to create a cube topology were each cube vertex has a SPIDER router with two nodes appended to it and additional three SPIDER connections are used to link the vertex to its neighbours. This is called a bristle cube. The remaining SPIDER connection is used to extend these 32 processors (4 processor x 8 vertices) by connecting it to other cubes. The Origin 2000 uses a directory-based cache coherence protocol similar to the Stanford DASH protocol [LLT+93]. The Origin 2000 is one of the most innovative architectures and is the setting of many recent research papers (e.g. [JS98] [BW97]).

The Cray T3E is a NUMA massively parallel multicomputer and one of the most recent machines by the Cray Research Corporation (now owned by SGI). The Cray T3E has from 6 to 2048 Alpha processors (some configurations use 600MHz processors) connected in a 3D cube interconnection topology. It has a 64-bit architecture with a maximum distributed globally addressable memory of 4 TBytes where remote memory access messages are automatically generated by the memory controller. Although remote memory can be addressed, this is not transparent to the programmer. Since there is no hardware mechanism to keep the Cray's processor caches consistent, they contain only local data. Each processor runs a copy of Cray's UNICOS microkernel on top of which language specific message-passing libraries support

parallel processing. The I/O system is distributed over a set of nodes on the surface of the cube.

NORMA – No Remote Memory Access. In a *NORMA machine* a processor is not able to access the memory of another processor. Consequently, all inter-processor communication has to be done by the explicit exchange of messages between processors and there is no global memory view. NORMA architectures are more cost-effective since they don't need all the hardware infrastructure associated with allowing multiple processor to access the same memory unit and therefore provide more raw processing power for a lower price at the expense of the system's programming model's friendliness.

The CM-5 [TMC90b] by the Thinking Machine Corporation is a MIMD machine, which contrasts with the earlier SIMD CM-1 and CM-2, a further sign of the general tendency of vendors to concentrate on MIMD architectures. The CM-5 is a private memory NORMA machine with a special type of processor interconnection called a fat tree. In a fat tree architecture, processors are the leaves of an interconnection tree where the bandwidth is bigger at the branches placed higher on the tree. This special interconnection reduces the bandwidth degradation implied by an increase of the distance between processors and it also enables bandwidth to remain constant as the number of processors grows. Each node in a CM-5 has a SPARC processor, a 64k cache and up to 32MB of local 64 bit-word memory. The CM-5 processors, which run a microkernel, are controlled by a control processor running UNIX, which serves as the user front end.

The Intel Paragon Supercomputer [Div95] is a 2D rectangular mesh of up to 4096 Intel i860 XP processors. This supercomputer with its 50Mhz 42 MIPS processors connected point-to-point at 200MByte/sec full duplex competed for a while with the CM-2 for the title of the "world's fastest supercomputer". The processor mesh is supported by I/O processors connected to hard disks and other I/O devices. The Intel Paragon runs the OSF/1 operating system, which is based on Mach.

In conclusion, it should be said that, after a period when very varied designs coexisted, parallel architectures are converging towards a generalized use of shared memory MIMD machines. There are calculus intensive domains were NORMA machines, such as the CRAY T3-E, are used but the advantages offered by the SMP architecture, such as the ease of programming, standard operating systems, use of standard hardware components and software portability, have overwhelmed the parallel computing market.

2.2 Parallel Operating Systems

There are several components in an operating system that can be parallelized. Most operating systems do not approach all of them and do not support parallel applications directly. Rather, parallelism is frequently exploited by some additional software layer such as a distributed file system, distributed shared

memory support or libraries and services that support particular parallel programming languages while the operating system manages concurrent task execution.

The convergence in parallel computer architectures has been accompanied by a reduction in the diversity of operating systems running on them. The current situation is that most commercially available machines run a flavour of the UNIX OS (Digital UNIX, IBM AIX, HP UX, Sun Solaris, Linux). Others run a UNIX based microkernel with reduced functionality to optimise the use of the CPU, such as Cray Research's UNICOS. Finally, a number of shared memory MIMD machines run Microsoft Windows NT (soon to be superseded by the high end variant of Windows 2000).

There are a number of core aspects to the characterization of a parallel computer operating system: general features such as the degrees of coordination, coupling and transparency; and more particular aspects such as the type of process management, inter-process communication, parallelism and synchronization and the programming model.

2.2.1 Coordination. The type of coordination among processors in parallel operating systems is a distinguishing characteristic, which conditions how applications can exploit the available computational nodes. Furthermore, application parallelism and operating system parallelism are two distinct issues: While application concurrency can be obtained through operating system mechanisms or by a higher layer of software, concurrency in the execution of the operating system is highly dependant on the type of processor coordination imposed by the operating system and the machine's architecture. There are three basic approaches to coordinating processors:

- Separate supervisor - In a *separate supervisor parallel machine* each node runs its own copy of the operating system. Parallel processing is achieved via a common process management mechanism allowing a processor to create processes and/or threads on remote machines. For example, in a multicomputer like the Cray T3E, each node runs its own independent copy of the operating system. Parallel processing is enabled by a concurrent programming infrastructure such as an MPI library (see Section 4.) whereas I/O is performed by explicit requests to dedicated nodes. Having a front-end that manages I/O and that dispatches jobs to a back-end set of processors is the main motivation for separate supervisor operating system.
- Master-slave - A *master-slave parallel operating system* architecture assumes that the operating system will always be executed on the same processor and that this processor will control all shared resources and in particular process management. A case of master-slave operating system, as we saw above, is the CM-5. This type of coordination is particularly adapted to single purpose machines running applications that can be broken into similar concurrent tasks. In these scenarios, central control may be maintained without any penalty to the other processors' performance since all processors tend to be beginning and ending tasks simultaneously.

– Symmetric - Symmetric OSs are the most common configuration currently. In a *symmetric parallel OS*, any processor can execute the operating system kernel. This leads to concurrent accesses to operating system components and requires careful synchronization. In Section 3. we will discuss the reasons for the popularity of this type of coordination.

2.2.2 Coupling and transparency. Another important characteristic in parallel operating systems is the system's degree of coupling. Just as in the case of hardware where we can speak of loosely coupled (e.g. network of workstations) and highly coupled (e.g. vector parallel multiprocessors) architectures, parallel OS can be meaningfully separated into loosely coupled and tightly coupled operating systems.

Many current distributed operating systems have a highly modular architecture. There is therefore a wide spectrum of distribution and parallelism in different operating systems. To see how influential coupling is in forming the abstraction presented by a system, consider the following extreme examples: on the highly coupled end a special purpose vector computer dedicated to one parallel application, e.g. weather forecasting, with master-slave coordination and a parallel file system, and on the loosely coupled end a network of workstations with shared resources (printer, file server) running each their own application and maybe sharing some client-server applications.

Within the spectrum of operating system coupling there are three landmark types:

– *Network Operating Systems* - These are implementations of a loosely coupled operating systems on top of loosely coupled hardware. Network operating systems are the software that supports the use of a network of machines and provide users that are aware of using a set of computers, with facilities designed to ease the use of remote resources located over the network. These resources are made available as services and might be printers, processors, file systems or other devices. Some resources, of which dedicated hardware devices such as printers, tape drives, etc... are the classical example, are connected to and managed by a particular machine and are made available to other machines in the network via a service or daemon. Other resources, such as disks and memory, can be organized into true distributed systems, which are seamlessly used by all machines. Examples of basic services available on a network operating system are starting a shell session on a remote computer (remote login), running a program on a remote machine (remote execution) and transferring files to and from remote machines (remote file transfer).
– *Distributed Operating Systems* - True distributed operating systems correspond to the concept of highly coupled software using loosely coupled hardware. Distributed operating systems aim at giving the user the possibility of transparently using a virtual uniprocessor. This requires having an adequate distribution of all the layers of the operating system and providing a global unified view over process management, file system and inter-

process communication, thereby allowing applications to perform transparent migration of data, computations and/or processes. Distributed file systems (e.g. AFS, NFS) and distributed shared memories (e.g. TreadMarks [KDC+94], Midway[BZ91], DiSoM [GC93]) are the most common supports for migrating data. Remote procedure call (e.g. Sun RPC, Java RMI, Corba) mechanisms are used for migration of computations. Process migration is a less common feature. However, some experimental platforms have supported it [Nut94], e.g. Sprite, Emerald.

– *Multiprocessor Timesharing OS* - This case represents the most common configuration of highly coupled software on top of highly coupled software. A multiprocessor is seen by the user as a powerful uniprocessor since it hides away the presence of multiple processor and an interconnection network. We will discuss some design issues of SMP operating systems in the following section.

2.2.3 HW vs. SW. As we mentioned a parallel operating system provides users with an abstract computational model over the computer architecture. It is worthwhile showing that this view can be achieved by the computer's parallel hardware architecture or by a software layer that unifies a network of processors or computers. In fact, there are implementations of every computational model both in hardware or software systems:

– The hardware version of the shared memory model is represented by symmetric multiprocessors whereas the software version is achieved by unifying the memory of a set of machines by means of a distributed shared memory layer.
– In the case of the distributed memory model there are multicomputer architectures where accesses to local and remote data are explicitly different and, as we saw, have different costs. The equivalent software abstractions are explicit message-passing inter-process communication mechanisms and programming languages.
– Finally, the SIMD computation model of massively parallel computers is mimicked by software through data parallel programming. Data parallelism is a style of programming geared towards applying parallelism to large data sets, by distributing data over the available processors in a "divide and conquer" mode. An example of a data parallel programming language is HPF (High Performance Fortran).

2.2.4 Protection. Parallel computers, being multi-processing environments, require that the operating system provide protection among processes and between processes and the operating system so that erroneous or malicious programs are not able to access resources belonging to other processes. Protection is the access control barrier, which all programs must pass before accessing operating system resources. Dual mode operation is the most common protection mechanism in operating systems. It requires that all operations that interfere with the computer's resources and their management

is performed under operating system control in what is called protected or kernel mode (in opposition to unprotected or user mode). The operations that must be performed in kernel mode are made available to applications as an operating system API. A program wishing to enter kernel mode calls one of these system functions via an interruption mechanism, whose hardware implementation varies among different processors. This allows the operating system to verify the validity and authorization of the request and to execute it safely and correctly using kernel functions, which are trusted and well-behaved.

3. Operating Systems for Symmetric Multiprocessors

Symmetric Multiprocessors (SMP) are currently the predominant type of parallel computers. The root of their popularity is the ease with which both operating systems and applications can be ported onto them. Applications programmed for uniprocessors can be run on an SMP unchanged, while porting a multitasking operating system onto an SMP requires minor changes. Therefore, no other parallel architecture is so flexible running both parallel and commodity applications on top of generalist operating systems (UNIX, NT). The existence of several processors raises scalability and concurrency issues, e.g. hardware support for cache coherence (see Section 2.), for the initialisation of multiple processors and the optimisation of processor interconnection. However, regarding the operating system, the specific aspects to be considered in a SMP are process synchronization and scheduling.

3.1 Process Management

A process is a program's execution context. It is a set of data structures that specify all the information needed to execute a program: its execution state (global variables), the resources it is using (files, processes, synchronization variables), its security identity and accounting information.

There has been an evolution in the type of tasks executed by computers. The state description of a traditional process contains much more information than what is needed to describe a simple flow of execution and therefore commuting between processes can be a considerably costly operation. Hence, most operating systems have extended processes with lightweight processes (or threads) that represent multiple flows of execution within the process addressing space. A thread needs to maintain much less state information than a process, typically it is described by its stack, CPU state and a pointer to the function it is executing, since it shares process resources and parts of the context with other threads within the same address space. The advantage of threads is that due to their low creation and context switch performance penalties they become a very interesting way to structure computer programs and to exploit parallelism.

There are operating systems, which provide kernel threads (Mach, Sun Solaris), while others implement threads as a user level library. In particular, user threads, i.e. threads implemented as a user level library on top of a single process, have very low commuting costs. The major disadvantage of user-level threads is the fact that the assignment of tasks to threads at a user-level is not seen by the operating system which continues to schedule processes. Hence, if a thread in a process is blocked on a system call (as is the case in many UNIX calls), all other threads in the same process become blocked too.

Therefore, support for kernel threads has gradually appeared in conventional computer operating systems. Kernel threads overcame the above disadvantages of user level threads and increased the performance of the system.

A crucial aspect of thread performance is the locking policy within the thread creation and switching functions. Naive implementations of these features, guarantee their execution as critical sections, by protecting the thread package with a single lock. More efficient implementations use finer and more careful locking implementations.

3.2 Scheduling

Scheduling is the activity of assigning processor time to the active tasks (processes or threads) in a computer and of commuting them. Although sophisticated schedulers implement complex process state machines there is a basic set of process (or thread) states that can be found in any operating system: A process can be running, ready or blocked. A running process is currently executing on one of the machine's processor whereas a ready process is able to be executed but has no available processor. A blocked process is unable to run because it is waiting for some event, typically the completion of an I/O operation or the occurrence of a synchronization event.

The run queue of an operating systems is a data structure in the kernel that contains all processes that are ready to be executed and await an available processor. Typically processes are enqueued in a run queue and are scheduled to the earliest available processor. The scheduler executes an algorithm, which places the running task in the run queue, selects a ready task from the run queue and starts its execution. At operating system level, what unites the processors in an SMP is the existence of a single run queue. Whenever a process is to be removed from a processor, this processor runs the scheduling algorithm to select a new running process. Since the scheduler can be executed by any processor, it must be run as a critical section, to avoid the simultaneous choice of a process by two separate executions of the scheduler. Additionally, most SMP provide methods to allow a process to run only on a given processor.

The simplest scheduling algorithm is first-come first-served. In this case, processes are ordered according to their creation order and are scheduled in that order typically by means of first-in first-out (FIFO) queue. A slight

improvement in overall performance can be achieved by scheduling jobs according to their expected duration. Estimating this duration can be a tricky issue.

Priority scheduling is the approach taken in most modern operating system. Each job is assigned a priority and scheduled according to that value. Jobs with equal priorities are scheduled in FIFO order.

Scheduling can be preemptive or non-preemptive. In preemptive scheduling, a running job can be removed from the processor if another job with a higher priority becomes active. The advantage of non-preemption is simplicity: by avoiding to preempt a job, the operating system knows that the process has left the processor correctly without leaving its data structures or the resources it uses corrupted. The disadvantage of non-preemption is the slowness of reaction to hardware events, which cannot be handled until the running process has relinquished the processor.

There are a number of measures of the quality of an OS' scheduling algorithm. The first is CPU utilization. We would like to optimise the amount of time that the machine's processors are busy. CPU utilization is not however a user oriented metric. Users are more interested in other measures, notably the amount of time a job needs to run and, in an interactive system, how responsive the machine is, i.e. whether a user-initiated request yields results quickly or not. A measure that translates all these concerns well is the average waiting time of a process, i.e. the amount of time a process spends in the waiting queue.

The interested reader may find more about scheduling strategies in Chapter VI.

3.3 Process Synchronization

In an operating system with multiple threads of execution, accesses to shared data structures require synchronization in order to guarantee that these are used as critical sections. Gaining permission to enter a critical section, which involves testing whether another process has already an exclusive access to this section, and if not, locking all other processes out of it, has to be performed atomically. This requires hardware mechanisms to guarantee that this sequence of actions is not preempted. Mutual exclusion in critical sections is achieved by requiring that a lock is taken before entering the critical section and by releasing it after exiting the section.

The two most common techniques for manipulating locks are the test-and-set and swap operations. Test-and-set atomically sets a lock to its locked state and returns the previous state of that lock to the caller process. If the lock was not set, it is set by the test and set call. If it was locked, the calling process will have to wait until it is unlocked.

The swap operation exchanges the contents of two memory locations. Swapping the location of a lock with a memory location containing the value corresponding to its locked state is equivalent to a test-and-set operation.

Synchronizing processes or threads on a multiprocessor poses an additional requirement. Having a busy-waiting synchronization mechanism, also known as a spin lock, in which a process is constantly consuming bandwidth on the computer's interconnection to test whether the lock is available or not, as in the case of test-and-set and swap, is inefficient. Nevertheless, this is how synchronization is implemented at the hardware level in most SMP, e.g. those using Intel processors. An alternative to busy-waiting is suspending processes which fail to obtain a lock and sending interrupt calls to all suspended processes when the lock is available again. Another synchronization primitive which avoids spin a lock are semaphores. Semaphores consist of a lock and a queue, which is manipulated in mutual exclusion, containing all the processes that failed to acquire the lock. When the lock is freed, it is given to one of the enqueued processes, which is sent an interrupt call. There are other variations of synchronization primitives such as monitors [Hoa74], eventcount and sequencers [RK79], guarded commands [Dij75] and others but the examples above illustrate clearly the issues involved in SMP process synchronization.

3.4 Examples of SMP Operating Systems

Currently, the most important examples of operating systems for parallel machines are UNIX and Windows NT running on top of the most ubiquitous multi-processor machines, which are symmetric multiprocessors.

3.4.1 UNIX. UNIX is one the most popular and certainly the most influential operating system ever built. Its development began in 1969 and from then on countless versions of UNIX have been created by most major computer companies (AT&T, IBM, Sun, Microsoft, DEC, HP). It runs on almost all types of computers from personal computers to supercomputers and will continue to be a major player in operating system practice and research. There are currently around 20 different flavours of UNIX being released. This diversity led to various efforts to standardize UNIX and so there is an ongoing effort by the Open Group, supported by all major UNIX vendors, to create a unified interface for all UNIX flavours.

Architecture. UNIX was designed as a time-sharing operating system where simplicity and portability are fundamental. UNIX allows for multiple processes that can be created asynchronously and that are scheduled according to a simple priority algorithm. Input and output in UNIX is as similar as possible among files, devices and inter-process communication mechanisms. The file system has a hierarchical structure and includes the use of demountable volumes. UNIX was initially a very compact operating system. As technology progressed, there have been several additions to it such as support for graphical interfaces, networking and SMP that have increased its size considerably, but UNIX has always retained its basic design traits.

UNIX is a dual mode operating system: the kernel is composed of a set of components that are placed between the system call interface provided to the user and the kernel's interface to the computer's hardware. The UNIX kernel is composed of:

- Scheduler - The scheduler is the algorithm which decides which process is run next (see *Process management* below). The scheduling algorithm is executed by the process occupying the processor when it is about to relinquish it. The context switch between processes is performed by the *swapper* process, which is one of the two processes constantly running (the other one is the *init* process which is the parent process of all other processes on the machine).
- File system - A file in UNIX is a sequence of bytes, and the operating system does not recognize any additional structure in it. UNIX organizes files in a hierarchical structure that is captured in a metadata structure called the *inode* table. Each entry in the *inode* table represents a file. Hardware devices are represented in the UNIX file system in the */dev* directory which contributes to the interface standardization of UNIX kernel components. This way, devices can be read and written just like files. Files can be known in one or more directories by several names, which are called links. Links can point directly to a file within the same file system (hard links) or simply refer to the name of another file (symbolic links). UNIX supports mount points, i.e. any file can be a reference to another physical file system. *inodes* have a bit for indicating whether they are mount points or not. If a file that is a mount point is accessed, a table with the equivalence between filenames and mounted file system, the mount table, is scanned to find the corresponding file system. UNIX filenames are sequences of filenames separated by '/' characters. Since UNIX supports mount points and symbolic links, filename parsing has to be done on a name by name basis. This is required because any filename along the path to the file being ultimately referred to can be a mount point or symbolic link and indirect the path to the wanted file.
- Inter-process communication (IPC) mechanisms - Pipes, sockets and, in more recent versions of UNIX, shared memory are the mechanisms that allow processes to exchange data (see below).
- Signals - Signals are a mechanism for handling exceptional conditions. There are 20 different signals that inform a process of events such as arithmetical overflow, invalid system calls, process termination, terminal interrupts and others. A process checks for signals sent to it when it leaves the kernel mode after ending the execution of a system call or when it is interrupted. With the exception of signals to stop or kill a process, a process can react to signals as it wishes by defining a function, called a signal handler, which is executed the next time it receives a particular signal.
- Memory management - Initially, memory management in UNIX was based on a swapping mechanism. Current versions of UNIX resort also to paging

to manage memory. In non-paged UNIX, processes were kept in a contiguous address space. To decide where to create room for a process to be brought from disk into memory (swapped-in), the swapper process used a criterion based on the amount of idle time or the age of processes in memory to choose which one to send to disk (swap-out). Conversely, the amount of time a process has been swapped out was the criterion to consider it for swap-in.

Paging has two great advantages: it eliminates external memory fragmentation and it eliminates the need to have complete processes in memory, since a process can request additional pages as it goes along. When a process requests a page and there isn't a page frame in memory to place the page, another page will have to be removed from the memory and written to disk.

The target of any page replacement algorithm is to replace the page that will not be used for the longest time. This ideal criterion cannot be applied because it requires knowledge of the future. An approximation to this algorithm is the least-recently-used (LRU) algorithm which removes from memory the page that hasn't been accessed for the longest time. However, LRU requires hardware support to be efficiently implemented. Since most systems provide a reference bit on each page that is set every time the page is accessed, this can be used to implement a related page replacement algorithm, one which can be found in some implementations of UNIX, e.g. 4.3BSD, the second chance algorithm. It is a modified FIFO algorithm. When a page is selected for replacement its reference bit is first checked. If this is set, it is then unset but the page gets a second chance. If that page is later found with its reference bit unset it will be replaced.

Several times per second, there is a check to see if it is necessary to run the page replacement algorithm. If the number of free page frames falls beneath a threshold, the *pagedaemon* runs the page replacement algorithm.

In conclusion, it should be mentioned that there is an interaction between swapping, paging and scheduling: as a process looses priority, accesses to its pages become more infrequent, pages are more likely to be paged out and the process risks being swapped out to disk.

Process management. In the initial versions of UNIX, the only execution context that existed were processes. Later several thread libraries were developed (e.g. POSIX threads). Currently, there are UNIX releases that include kernel threads (e.g. Sun's Solaris 2) and the corresponding kernel interfaces to manipulate them.

In UNIX, a process is created using the *fork* system call, which generates a process identifier (or *pid*). A process that executes the *fork* system call receives a return value, which is the *pid* of the new process, i.e. its child process, whereas the child process itself receives a return code equal to zero. As a consequence of this process creation mechanism, processes in UNIX are organized as a process tree. A process can also replace the program it is

executing with another one by means of the *execve* system call. Parent and children processes can be synchronized by using the *exit* and *wait* calls. A child process can terminate its execution when it calls the *exit* function and the parent process can synchronize itself with the child process by calling *wait*, thereby blocking its execution until the child process terminates.

Scheduling in UNIX is performed using a simple dynamic priority algorithm, which benefits interactive programs and where larger numbers indicate a lower priority. Process priority is calculated using:

$Priority = Processor\_time + Base\_priority[+nice]$

The nice factor is available for a processor to give other processes a higher priority. For each quantum that a process isn't executed, it's priority improves via:

$Processor\_time = Processor\_time/2$

Processes are assigned a CPU slot, called a quantum, which expires by means of a kernel timeout calling the scheduler. Processes cannot be preempted while executing within a kernel. They relinquish the processor either because they blocked waiting for I/O or because their quantum has expired.

Inter-process communication. The main inter-process communication mechanism in UNIX are pipes. A pipe is created by the *pipe* system call, which establishes a reliable unidirectional byte stream between two processes. A pipe has a fixed size and therefore blocks writer processes trying to exceed their size. Pipes are usually implemented as files, although they do not have a name. However, since pipes tend to be quite small, they are seldom written to disk. Instead, they are manipulated in the block cache, thereby remaining quite efficient. Pipes are frequently used in UNIX as a powerful tool for concatenating the execution of UNIX utilities. UNIX shell languages use a vertical bar to indicate that a program's output should be used as another program's input, e.g. listing a directory and then printing it ($ls \mid lpr$).

Another UNIX IPC mechanism are sockets. A socket is a communication endpoint that provides a generic interface to pipes and to networking facilities. A socket has a domain (UNIX, Internet or Xerox Network Services), which determines its address format. A UNIX socket address is a file name while an Internet socket uses an Internet address. There are several types of sockets. Datagram sockets, supported on the Internet UDP protocol, exchange packets without guarantees of delivery, duplication or ordering. Reliable delivered message sockets should implement reliable datagram communication but they are not currently supported. Stream sockets establish a data stream between two sockets, which is duplex, reliable and sequenced. Raw sockets allow direct access to the protocols that support other socket types, e.g. to access the IP or Ethernet protocols in the Internet domain. Sequenced packet sockets are used in the Xerox AF_NS protocol and are equivalent to stream sockets with the addition of record boundaries.

A socket is created with a call to *socket*, which returns a socket descriptor. If the socket is a server socket, it must be bound to a name (*bind* system call)

so that client sockets can refer to it. Then, the socket informs the kernel that it is ready to accept connections (*listen*) and waits for incoming connections (*accept*). A client socket that wishes to connect to a server socket is also created with a *socket* call and establishes a connection to a server socket. Its name is known after performing a *connect* call. Once the connection is established, data can be exchanged using ordinary *read* and *write* calls.

A system call complementary to UNIX IPC mechanisms is the *select* call. It is used by a process that wishes to multiplex communication on several files or socket descriptors. A process passes a group of descriptors into the select call that returns the first on which activity is detected.

Shared memory support has been introduced in several recent UNIX versions, e.g. Solaris 2. Shared memory mechanisms allow processes to declare shared memory regions (*shmget* system call) that can then be referred to via an identifier. This identifier allows processes to attach (*shmat*) and detach (*shmdt*) a shared memory region to and from its addressing space. Alternatively, shared memory can be achieved via memory-mapped files. A user can choose to map a file (*mmap*/*munmap*) onto a memory region and, if the file is mapped in a shared mode, other processes can write to that memory region thereby communicating among them.

Internal parallelism. In a symmetric multiprocessor system, all processors may execute the operating system kernel. This leads to synchronization problems in the access to the kernel data structures. The granularity at which these data structures are locked is highly implementation dependent. If an operating system has a high locking cost it might prefer to lock at a higher granularity but lock less often. Other systems with a lower locking cost might prefer to have complex fine-grained locking algorithms thereby reducing the probability of having processes blocked on kernel locks to a minimum. Another necessary adaptation besides locking of kernel data structures is to modify the kernel data structures to reflect the fact that there is a set of processors.

3.4.2 Microsoft Windows NT. *Windows NT* [Cus93] is the high end of Microsoft Corporation's operating systems range. It can be executed on several different architectures (Intel x86, MIPS, Alpha AXP, PowerPC) with up to 32 processors. Windows NT can also emulate several OS environments (Win32, OS/2 or POSIX) although the primary environment is Win32. NT is a 32-bit operating system with separate per-process address spaces. Scheduling is thread based and uses a preemptive multiqueue algorithm. NT has an asynchronous I/O subsystem and supports several file systems: FAT, high-performance file system and the native NT file system (NTFS). It has integrated networking capabilities which support 5 different transport protocols: NetBeui, TCP/IP, IPX/SPX, AppleTalk and DLC.

Architecture. Windows NT is a dual mode operating system with user-level and kernel components. It should be noted that not the whole kernel is ex-

ecuted in kernel mode. The Windows NT components that are executed in kernel mode are:

- Executive - The executive is the upper layer of the operating system and provides generic operating system services for managing processes, threads and memory, performing I/O, IPC and ensuring security.
- Device Drivers - Device drivers provide the executive with a uniform interface to access devices independently of the hardware architecture the manufacturer uses for the device.
- Kernel - The kernel implements processor dependent functions and related functions such as thread scheduling and context switching, exception and interrupt dispatching and OS synchronization primitives
- Hardware Abstraction Layer (HAL) - All these components are layered on top of a hardware abstraction layer. This layer hides hardware specific details, such as I/O interfaces and interrupt controllers, from the NT executive thereby enhancing its portability. For example, all cache coherency and flushing in a SMP is hidden beneath this level.

The modules that are executed in user mode are:

- System & Service Processes - These operating system processes provide several services such as session control, logon management, RPC and event logging.
- User Applications
- Environment Subsystems - The environment subsystems use the generic operating system services provided by the executive to give applications the interface of a particular operating system. NT includes environment subsystems for MS-DOS, 16 bit Windows, POSIX, OS/2 and for its main environment, the Win32 API.

In NT, operating system resources are represented by objects. Kernel objects include processes and threads, file system components (file handles, logs), concurrency control mechanisms (semaphores, mutexes, waitable timers) and inter-process communication resources (pipes, mailboxes, communication devices). In fact, most of the objects just mentioned are NT executive interfaces to low-level kernel objects. The other two types of Windows NT OS objects are the user and graphical interface objects which compose the graphical user interface and are private to the Win32 environment.

Calls to the kernel are made via a protected mechanism that causes an exception or interrupt. Once the system call is done, execution returns to user-level by dismissing the interrupt. These are calls to the environment subsystems libraries. The real calls to the kernel libraries are not visible at user-level.

Security in NT is based on two mechanisms: processes, files, printers and other resources have an access control list and active OS objects (processes and threads) have security tokens. These tokens identify the user on behalf of which the process is executing, what privileges that user possesses and can

therefore be used to check access control when an application tries to access a resource.

Process management. A Windows NT process is a data structure where the resources held by an execution of a program are managed and accounted for. This means that a process keeps track of the virtual addressing space and the corresponding working set its program is using. Furthermore an NT process includes a security token identifying its permissions, a table containing the kernel objects it is using and a list of threads. A process' handle table is placed in the system's address space and therefore protected from user-level "tampering".

In NT, a process does not represent a flow of execution. Program execution is always performed by a thread. In order to trace an execution a thread is composed of a unique identifier, a set of registers containing processor state, a stack for use in kernel mode and a stack for use in user mode, a pointer to the function it is executing, a security access token and its current access mode.

The Windows NT kernel has a multi-queue scheduler. It has 32 different priority levels. The upper 15 are reserved for real time tasks. Scheduling is preemptive and strictly priority driven. There is no guaranteed execution period before preemption and threads can be preempted in kernel mode. Within a priority level, scheduling is time-sliced and round robin. If a thread is preempted it goes back to the head of its priority level queue. If it exhausts its CPU quantum, it goes back to the tail of that queue.

Regarding multiprocessor scheduling, each task has an ideal processor, which is assigned, at the thread's creation, in a round-robin fashion among all processors. When a thread is ready, and more than one processor is available, the ideal processor is chosen. Otherwise, the last processor where the thread ran is used. Finally, if the thread has not been waiting for too long, the scheduler checks to see if the next thread in the ready state can run on its ideal processor. If it can, this second thread is chosen. A thread can run on any processor unless an affinity mask is defined. An affinity mask is a sequence of bits where each represents a processor and which can be set to restrict the processors where a thread is allowed to execute. However, this this forced thread placement, called hard affinity, may lead to a reduced execution of the thread.

Inter-process communication. Windows NT provides several different inter-process communication (IPC) mechanisms:

Windows sockets. The Windows Sockets API provides a standard Windows interface to several communication protocols with different addressing schemes, such as TCP/IP and IPX. The Windows Sockets API was developed to accomplish two things. One was to migrate the sockets interface, developed for UNIX BSD in the early 1980s, into the Windows NT environments, and the other was to establish a new standard interface capable of supporting emerging network capabilities such as real time communications

and QoS (quality of service) guarantees. The Windows Sockets, currently in their version 2, are an interface for networking, not a protocol. As they do not implement a particular protocol, the Windows Sockets do not affect the data being transferred through them. The underlying transport protocol can be of any type. Besides the usual datagram and stream sockets, it can, for example, use a protocol designed for multimedia communications. The transport protocol and name providers are placed beneath the Windows Socket layer.

Local Procedure Calls. LPC is a message-passing facility provided by the NT executive. The LPC interface is similar to the standard RPC but it is optimised for communication between two processes on the same machine. LPC communication is based on ports. A port is a kernel object that can have two types: connection port and communication port. Connections are established by sending a connect request to another process' connection port. If the connection is accepted, communication is then established via two newly created ports, one on each process. LPC can be used to pass messages directly through the communication ports or to exchange pointers into memory regions shared by both communicating processes.

Remote Procedure Calls (RPC). Apart from the conventional inter-computer invocation procedure, the Windows NT RPC facility can use other IPC mechanisms to establish communications between the computers on which the client and the server portions of the application exist. If the client and server are on the same computer, the Local Procedure Call (LPC) mechanism can be used to transfer information between processes and subsystems. This makes RPC the most flexible and portable IPC choice in Windows NT.

Named pipes and mailslots. Named pipes provide connection-oriented messaging that allows applications to share memory over the network. Windows NT provides a special application programming interface (API) that increases security when using named pipes.

The mailslot implementation in Windows NT is a subset of the Microsoft OS/2 LAN Manager implementation. Windows NT implements only second-class mailslots. Second class mailslots provide connectionless messaging for broadcast messages. Delivery of the message is not guaranteed, though the delivery rate on most networks is high. It is most useful for identifying other computers or services on a network. The Computer Browser service under Windows NT uses mailslots. Named pipes and mailslots are actually implemented as file systems. As file systems, they share common functionality, such as security, with the other file systems. Local processes can use named pipes and mailslots without going through the networking components.

NetBIOS. NetBIOS is a standard programming interface in the PC environment for developing client-server applications. NetBIOS has been used as an IPC mechanism since the early 1980s. From a programming aspect, higher level interfaces such as named pipes and RPCs are superior in their

flexibility and portability. A NetBIOS client-server application can communicate over various protocols: NetBEUI protocol (NBF), NWLink NetBIOS (NWNBLink), and NetBIOS over TCP/IP (NetBT).

Synchronization. In Windows NT, the concept of having entities through which threads can synchronize their activities is integrated in the kernel object architecture. The synchronization objects, objects on which a thread can wait and be signalled, in the NT executive are: process, thread, file, event, event pair, semaphore, timer and mutant (seen at Win32 level as a mutex). Synchronization on these objects differs in the conditions needed for the object to signal the threads that are waiting on it. The circumstances in which synchronization objects signal waiting threads are: processes when their last thread terminates, threads when they terminate, files when an I/O operation finishes, events when a thread sets them, semaphores when their counter drops to zero, mutants when holding threads release them, timers when their set time expires. Synchronization is made visible to Win32 applications via a generic wait interface (*WaitForSingleObject* or *WaitForMultipleObjects*).

4. Operating Systems for NORMA environments

In Section 2. we showed that parallel architectures have been restricted mainly to symmetric multiprocessors, multicomputers, computer clusters and networks of workstations. There are some examples of true NORMA machines, notably the Cray T3E. However since networks of workstations and/or personal computers are the most challenging NORMA environments today, we chose to approach the problem of system level support for NORMA parallel programming to raise some of the problems that arise in the network of workstations environment. Naturally, most of those problems are also highly prominent in NORMA multiprocessors.

While in a multiprocessor every processor is exactly like every other in capability, resources, software and communication speed, in a computer network that is not the case. As a matter of fact, the computers in a network are most probably heterogeneous, i.e. they have different hardware and/or operating systems. More precisely, the heterogeneity aspects that must be considered, when exploiting a set of computers in a network, are the following: architecture, data format, computational speed, machine and network load.

Heterogeneity of architecture means that the computers in the network can be Intel personal computers, workstations, shared-memory multiprocessors, etc... which poses the problems of incompatible binary formats and different programming methodologies.

Data format heterogeneity is an important obstacle to parallel computing because it prevents computers from correctly interpreting the data exchanged among them.

The heterogeneity of computational speed may result in unused processing power because the same task can be accomplished much faster in a multiprocessor than in a personal computer, for example. Thus, the programmer must be careful at splitting the tasks among the computers in the network so that no computer sits idle.

The computers in the network have different usage patterns, which lead to different loads. The same reasoning applies to network load. This may lead to a situation in which a computer takes a lot of time to execute a simple task, because it is too loaded, or stays idle, because it is waiting for some message.

All these problems must be solved in order to take advantage of the benefits of parallel computing on networks of computers, which can be summarized as follows:

- low price given that the computers already exist;
- optimised performance by assigning the right task to the most appropriate computer (taking into account its architecture, operating system and load);
- the resources available can easily grow by simply connecting more computers to the network (possibly with more advanced technologies);
- programmers still use familiar tools such as in a single computer (editors, compilers, etc.)

4.1 Architectural Overview

Currently, stand-alone workstations and personal computers are very ubiquitous and almost always connected by means of a network. Each machine provides a reasonable amount of processing power and memory which remains unused for long periods of time. The challenge is to provide a platform that allows programmers to take advantage of such resources easily. Thus, the computational power of such computers can be applied to solve a variety of computationally intensive applications. In other words, this network-based approach can be effective in coupling several computers, resulting in a configuration that might be economically and technically difficult to achieve with supercomputer hardware.

On the other extreme of the spectrum of NORMA systems are of course multicomputers generally running separate supervisor microkernels. These machines derive their performance from high performance interconnection and from very compact and therefore very fast operating system. The parallelism is exploited not directly by the operating system but by some language level functionality. Since each node is a complete processor and not a simplified processing node as in SIMD supercomputers, the tasks it performs can be reasonably complex.

System level support for NORMA architectures involves providing mechanism to overcome the physical processor separation and to coordinate activities on processors running independent operating systems. This section

discusses some of these basic mechanisms: distributed process management, parallel file systems and distributed recovery.

4.2 Distributed Process Management

In contrast with the SMP discussed in the previous section that are generally used as glorified uniprocessor in the sense that they execute an assorted set of applications, NORMA systems are targeted at parallel applications. *Load balancing* is an important feature to increase the performance of many parallel applications. However, efficient load balancing, in particular receiver initiated load balancing protocols, where processors request tasks from others, requires an underlying support for process migration.

4.2.1 Process migration. *Process migration* [Nut94] is the ability to move an executing process from one processor to another. It is a very useful functionality although it is costly to execute and challenging to implement. Basically, migration consists of halting a process in a consistent state in the current processor, packing a complete description of its state, choosing a new processor for the process, shipping the state description to the destination processor and restarting it there. Process migration is most frequently motivated by the need to balance the load of a parallel application. However, it is also useful for reducing inter-process communication (processes communicating with increasing frequency should be moved to the same processor), reconfiguring a system for administrative reasons or exploiting a special capability specific of a particular processor. There are several systems that implemented process migration such as Sprite[Dou91], Condor [BLL92] or Accent [Zay87]. More recently, as migration capabilities where introduced in object oriented distributed and parallel systems, the interest in migration has shifted into object migration, or mobile objects as in Emerald[MV93] or COOL[Lea93]. However, the core difficulties remain the same: ensuring that the migrating entity's state is completely described and capturing, redirecting and replaying all interactions that would have taken place during the migration procedure.

4.2.2 Distributed recovery. Recovery of a computation in spite of node failures is achieved via checkpointing, which is the procedure of saving the state of an ongoing distributed computation so that it can be recovered from that intermediate state in case of failure. It should be noted that checkpointing allows a computation to recover from a system failure. However, if an execution fails due to an error induced by the process' execution itself, the error will repeat itself after recovery. Another type of failures not addressed by checkpointing are secondary storage failures. Hard disk failures have to be addressed by replication using mirrored disks or RAID (cf. Section 8.4 of Chapter V).

Checkpointing and process migration require very similar abilities from the system point of view since the complete state of a process must be saved

so that the processor (the current one in the case of checkpointing or another in the case of process migration) can be correctly loaded and restarted with the process' saved state. There are four basic approaches for checkpointing [EJW96]:

- Coordinated checkpointing - In coordinated checkpointing, the processes involved in a parallel execution execute their checkpoints simultaneously in order to guarantee that the saved system state is consistent. Thus, the processes must periodically cooperate in computing a consistent global checkpoint. Coordinated checkpoints were devised because simplistic protocols with lack of coordination tend to create a so-called domino effect. The domino effect is due to the existence of orphan messages. An orphan message appears when a process, after sending messages to other processes, needs to rollback to a previous checkpoint. In that case, there will be processes that have received messages, which, from the point of view of the failed process that has rolled back, haven't been sent. Suppose that in a group of processes each saves its state independently. Then, if a failure occurs and a checkpoint has to be chosen in order to rollback to it, it is difficult to find a checkpoint where no process has orphan messages. As a result the system will regress indefinitely trying to find one checkpoint that is consistent for all processors. Coordinated checkpointing basically eliminates the domino effect, given that processes always restart from the most recent global checkpointing. This coordination even allows for processes to have non-deterministic behaviours. Since all processors are restarted from the same moment in the computation, the new computation could take a different execution path. The main problem with this approach is that it requires the coordination of all the processes, i.e. process autonomy is sacrificed and there is a performance penalty to be paid when all processes have to be stopped in order to save a globally consistent state.
- Uncoordinated checkpointing - With this technique, processes create checkpoints asynchronously and independently of each other. Thus, this approach allows each process to decide independently when to take checkpoints. Given that there is no synchronization among computation, communication and checkpointing, this optimistic recovery technique can tolerate the failure of an arbitrary number of processors. For this reason, when failures are very rare, this technique yields better throughput and response time than other general recovery techniques.
  When a processor fails, processes have to find a set of previous checkpoints representing a consistent system state. The main problem with this approach is the domino effect, which may force processes to undo a large amount of work. In order to avoid the domino effect, all the messages received since the last checkpoint was created would have to be logged.
- Communication-induced checkpointing - This technique complements uncoordinated checkpoints and avoids the domino effect by ensuring that all processes verify a system-wide constraint before they accept a message

from the outside world. A number of such constraints that guarantee the avoidance of a domino effect have been identified and this method requires that any incoming message is accompanied by the necessary information to verify the constraint. If the constraint is not verified, then a checkpoint must be taken before passing the message on to the parallel application. An example of a constraint for communication induced logging is the one proposed in Programmer Transparent Coordination (PTC) [KYA96] which requires that each process take a checkpoint if the incoming message makes it depend on a checkpoint that it did not previously depend upon.

– Log-based rollback recovery - This approach requires that processes save their state when they perform a checkpoint and additionally save all data regarding interactions with the outside world, i.e. emission and reception of messages. This stops the domino effect because the message log bridges the gap between the checkpoint to which the failed process rolled back and the current state of the other processes. This technique can even include non-deterministic events in the process' execution if these are logged too. A common way to perform logging is to save each message that a process receives before passing it on to the application code.
During recovery the logged events are replayed exactly as they occurred before the failure.
The difference of this approach with respect to to the previous ones is that in both, coordinated and uncoordinated checkpointing, the system restarts the failed processes by restoring a consistent state. The recovery of a failed process is not necessarily identical to its pre-failure execution which complicates the interaction with the outside world. On the contrary, log-based rollback recovery does not have this problem since it recorded all its previous interactions with the outside world.
There are three variants of this approach: pessimistic logging, optimistic logging, and causal logging.
In pessimistic logging, the system logs information regarding each non-deterministic event before it can affect the computation, e.g. a message is not delivered until it is logged.
Optimistic logging takes a more relaxed but more risky approach. Non-deterministic events are not written to stable storage but saved in a faster volatile log and only periodically written in an asynchronous way to a stable storage. This is a more efficient approach since it avoids constantly waiting for costly writes to a stable storage. However, should a process fail, the volatile log will be lost and there is the possibility of thereby creating orphan messages and inducing a domino effect.
Causal logging uses the Lamport *happened-before* relation [Lam78], to establish which events caused a process' current state, and guarantee that all those events are either logged or available locally to the process.

4.3 Parallel File Systems

Parallel file systems address the performance problem faced by current network *client-server file systems*: the read and write bandwidth for a single file is limited by the performance of a single server (memory bandwidth, processor speed, etc.). Parallel applications suffer from the above-mentioned performance problem because they present I/O loads equivalent to traditional supercomputers, which are not handled by current file servers. If users resort to parallel computers in search of fast execution, the file systems of parallel computers must also aim at serving I/O request rapidly.

4.3.1 Striped file systems. Dividing a file among several disks enables an application to access that file quicker by issuing simultaneous requests to several disks [CK93]. This technique is called declustering. In particular, if the file blocks are divided in a round-robin way among the available disks it is called striping. In a striped file system, either individual files are striped across the servers (per-file striping), or all the data is striped, independently of the files to which they belong, across the servers (per-client striping). In the first case, only large files benefit from the striping. In the second case, given that each client forms its new data for all files into a sequential log, even small files benefit from the striping.

With per-client striping, servers are used efficiently, regardless of file sizes. Striping a large amount of writes allows them to be done in parallel. On the other hand, small writes will be batched.

4.3.2 Examples of parallel file systems. CFS (Concurrent File System) [Pie89] is one of the first commercial multiprocessor file systems. It was developed for the Intel iPSC and Touchstone Delta multiprocessors. The basic idea of CFS is to decluster files across several I/O processors, each one with one or more disks. Caching and prefetching are completely under the control of the file system; thus, the programmer has no way to influence it. The same happens with the clustering of a file across disks, i.e. it is not predictable by the programmer.

CFS constitutes the secondary storage of the Intel Paragon. The nodes use the CFS for high-speed simultaneous access to secondary storage. Files are manipulated using the standard UNIX system calls in either C or FORTRAN and can be accessed by using a file pointer common to all applications, which can be used in four sharing modes:

- Mode 0: Each node process has its own file pointer. It is useful for large files to be shared among the nodes.
- Mode 1: The computer nodes share a common file pointer, and I/O requests are serviced on a first-come-first-serve basis.
- Mode 2: Reads and writes are treated as global operations and a global synchronization is performed.
- Mode 3: A synchronous ordered mode is provided, but all write operations have to be of the same size.

The PFS (Parallel File System) [Roy93] is a striped file system which stripes files (but not directories or links) across the UNIX file systems that were assembled to create the file system. The size of each stripe, the stripe unit is determined by the system administrator.

Contrary to the previous two file systems, the file system for the nCUBE multiprocessor [PFD+89] allows the programmer to control many of its features. In particular, the nCUBE file system, which also uses file striping, allows the programmer to manipulate the striping unit size and distribution pattern.

IBM's Vesta file system [CFP+93], a striped file system for the SP-1 and SP-2 multiprocessors, allows the programmer to control the declustering of a file when it is created. It is possible to specify the number of disks, the record size, and the stripe-unit size. Vesta provides users with a single name space. Users can dynamically partition files into subfiles, by breaking a file into rows, columns, blocks, or more complex cyclic partitioning. Once partitioned, a file can be accessed in parallel from a number of different processes of a parallel application.

4.4 Popular Message-Passing Environments

4.4.1 PVM. *PVM* [GBD94] supports parallel computing on current computers by providing the abstraction of a parallel virtual machine which comprehends all the computers in a network. For this purpose, PVM handles message routing, data conversion and task scheduling among the participating computers.

The programmer develops his programs as a collection of cooperating tasks. Each task interacts with PVM by means of a library of standard interface routines that provide support for initiating and finishing tasks on remote computers, sending and receiving messages and synchronizing tasks.

To cope with heterogeneity, the PVM message-passing primitives are strongly typed, i.e. they require type information for buffering and transmission.

PVM allows the programmer to start or stop any task, or to add or delete any computer in the network from the parallel virtual machine. In addition, any process may communicate or synchronize with any other. PVM is structured around the following notions:

- The user with an application running on top of PVM can decide which computers will be used. This set of computers, a user-configured host pool, can change during the execution of an application, by simply adding or deleting computers from the pool. This dynamic behaviour is very useful for fault-tolerance and load-balancing.
- Application programmers can specify which tasks are to be run on which computers. This allows the explicit exploitation of special hardware or operating system capabilities.

- In PVM, the unit of parallelism is a task that often, but not necessarily, is a UNIX process. There is no process-to-processor mapping implied or enforced by PVM.
- Heterogeneity: Varied networks and applications are supported. In particular, PVM allows messages containing more than one data type to be exchanged by machines with different data representations.
- PVM uses native message-passing support on those computers in which the hardware provides optimised message primitives.

4.4.2 MPI. *MPI* [MPI95] stands for Message Passing Interface. The goal of MPI is to become a widely used standard for writing message-passing programs. It is a standard that defines both the syntax and semantics, of a message-passing library that can efficiently be implemented on a wide range of computers. The defined interface attempts to establish a practical, portable, efficient, and flexible standard for message-passing. The MPI standard provides MPI vendors with a clearly defined base set of routines that they can implement efficiently or, in some cases, provide hardware support for, thereby enhancing scalability.

MPI has been strongly influenced by work at the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, and p4.

Blocking communication is the standard communication mode in MPI. By default, a sender of a message is blocked until the receiver calls the appropriate function to receive the message. However, MPI does support non-blocking communication. Although MPI is a complex standard, six basic calls are enough to create an application:

- `MPI_Init` - initiate a MPI computation
- `MPI_Finalize` - terminate a computation
- `MPI_Comm_size` - determine the number of processes
- `MPI_Comm_rank` - determine the current process' identifier
- `MPI_Send` - send a message
- `MPI_Recv` - receive a message

MPI is not a complete software infrastructure that supports distributed computing. In particular, MPI provides neither process management, i.e. the ability to create remote tasks, nor support for I/O. So, MPI is a communications interface layer that will be built upon native facilities of the underlying hardware platform. (Certain data transfer operations may be implemented at a level close to hardware.) For example, MPI could be implemented on top of PVM.

5. Scalable Shared Memory Systems

Traditionally parallel computing has been based on the message-passing model. However there are a number of experiences with distributed shared

memory that have shown that there is not necessarily a great performance penalty to pay for this more amenable programming model. The great advantage of shared memory is that it hides the details of interprocessor communication from the programmer. Furthermore, this transparency allows for a number of performance-enhancing techniques, such as caching and data migration and replication, to be used without changes to application code.

Of course, the main advantage of shared memory, transparency, is also the major flaw, as pointed out by the advocates of message-passing: a programmer with application specific knowledge can better control the placement and exchange of data than an underlying system can do it in an implicit, automatic way. In this section, we present some of the most successful scalable shared memory systems.

5.1 Shared Memory MIMD Computers

5.1.1 DASH. The goal of the Stanford DASH [LLT+93] project was to explore large-scale shared-memory architecture with hardware directory-based cache coherency. DASH consists of a two dimensional mesh of clusters. Each cluster contains four processors (MIPS R3000 with two levels of cache), up to 28 MB of main memory, a directory controller, which manages the memory metadata, and a reply controller, which is a board that handles all requests issued by the cluster. Within each cluster cache consistency is maintained with a snoopy bus protocol. Between clusters, cache consistency is guaranteed by a directory-based protocol. To enforce this protocol a full-map scheme is implemented. Each cluster has a complete map of the memory indicating whether a memory location is valid on the local memory or if the most recent value is held at a remote cluster. Requests for memory locations that are locally invalid are sent to the remote cluster, which replies to the requesting cluster. The DASH prototype is composed of 16 clusters with a total of 64 processors. Due to the various caching levels, locality is a relevant factor for the performance of DASH whose performance degrades as application working sets increase.

5.1.2 MIT Alewife. The MIT Alewife's architecture [ACJ+91] is similar to that of the Stanford DASH. It is composed by a 2D mesh of nodes, each with a processor (a modified SPARC called Sparcle), memory and a communication switch. However, the Alewife uses some interesting techniques to improve performance. The Sparcle processor is able to switch quickly context between threads. So when a thread accesses data that is not available locally, the processor, using independent register sets, quickly switches to another thread. Another interesting feature of the Alewife is its LimitLESS cache coherence technique. LimitLESS emulates a directory scheme similar to that of the DASH but it uses a very small map of cache line descriptors. Whenever a cache line is requested that is outside that limited set, the request is handled by a software trap. The leads to a much smaller performance degradation

than should be expected and has the obvious advantage of reducing memory occupation.

5.2 Distributed Shared Memory

Software distributed shared memory (DSM) systems aim at capturing the growing computational power of workstations and personal computers and the improvements in the networking infrastructure to create a parallel programming environment on top of them. We present two particularly successful implementations of software DSM: TreadMarks and Shasta.

5.2.1 TreadMarks. TreadMarks [KDC+94] is a DSM platform developed by Rice University, which uses lazy release consistency as its memory consistency model. A memory consistency model defines under which circumstances memory is updated in a DSM system. The release consistency model depends on the existence of synchronization variables whose accesses are divided into *acquire* and *release*. The release consistency model requires that accesses to regular variables are followed by the *release* of a synchronization variable which must be *acquired* by any other process before accessing the regular variables again. In particular, the lazy release consistency model postpones the propagation of memory updates performed by a process until another process acquires the corresponding synchronization variable.

Another fundamental option in DSM is deciding how to detect that applications are attempting to access memory. If the DSM layer wants to do the detection automatically by using memory page faults then the unit of consistency, i.e., the smallest amount of memory that can be updated in a consistency operation will be a memory page. Alternatively, smaller consistency units can be used but in this case the system can no longer rely on page faults and changes have to be made to the application code to indicate where data accesses are going to occur.

The problem with big consistency units is false sharing. False sharing happens when two processes are accessing different regions of a common consistency unit. In this case, they will be communicating frequently to exchange updates of that unit although in fact they are not sharing data at all.

The designers of TreadMarks chose the page-based approach. However they use a technique called lazy diff creation for reducing the amount of data exchanged to update a memory page. When an application first accesses a page TreadMarks creates a copy of the page (a twin). When another process requests an update of the page, a record of the modifications made to the page (a diff) is created by comparing the two twin pages, the initial and the current one. It is this diff that is sent to update the requesting processes page. TreadMarks has been used for several applications, such as genetic research where it was particularly successful.

5.2.2 Shasta. Shasta [SG97] is a software DSM system that runs on AlphaServer SMPs connected by a Memory Channel interconnection. The Memory Channel is a memory-mapped network that allows a process to transmit data to a remote process without any operating system overhead via a simple store into a mapped page. Processes check for incoming memory transfers by polling a variable.

Shasta uses variable sized memory units, which are multiples of 64 bytes. Memory units are managed using a per-processor table where the state of each block is kept. Additionally, each processor knows for each of its local memory units, the ones initially placed in its memory, which processors are holding those memory units.

Shasta implements its memory consistency protocol by instrumenting the application code, i.e. altering the binary code of the compiled applications. Each access to application data is bracketed in code that enforces memory consistency. The added code doubles the size of applications. However using a set of optimisations, such as instrumenting sequences of contiguous memory accesses as a single access, this overhead drops to an average of 20 %. Shasta is a paradigmatic example of a non-intrusive transition of a uniprocessor application to a shared memory platform: the designers of Shasta were able to execute the Oracle database transparently on top of Shasta.

References

[ACJ+91] Agarwal, A., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B., Maa, G., Nussbaum, D., *The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor, Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991.

[Bac93] Bacon, J., *Concurrent Systems, An Integrated Approach to Operating Systems, Database, and Distributed System*, Addison-Wesley, 1993.

[BZ91] Bershad, B.N., Zekauskas, M., Midway, J., Shared memory parallel programming with entry consistency for distributed memory multiprocessors, Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, 1991.

[Bla90] Blank, T., The MasPar MP-1 architecture, *Proc. 35th IEEE Computer Society International Conference (COMPCON)*, 1990, 20-24.

[BLL92] Bricker, A., Litzkow, M., Livny, M., Condor technical summary, Technical Report, Computer Sciences Department, University of Wisconsin-Madison, 1992.

[BW97] Brooks III, E.D., Warren, K.H., A study of performance on SMP and distributed memory architectures using a shared memory programming model, *SC97: High Performance Networking and Computing*, San Jose, 1997.

[Cha98] Charlesworth, A., Starfire: Extending the SMP envelope, *IEEE Micro* **18**, 1998.

[CFP+93] Corbett, P.F., Feitelson, D.G., Prost, J.P., Johnson-Baylor, S.B., Parallel access to files in the vesta file system, *Proc. Supercomputing '93*, 1993, 472-481.

[CK93] Cormen, T.H., Kotz, D., Integrating theory and practice in parallel file systems, *Proc. the 1993 DAGS/PC Symposium*, 1993, 64-74.

[Cus93] Custer, H., *Inside Windows NT*, Microsoft Press, Washington, 1993.

[Dij75] Dijkstra, E.W., Guarded commands, nondeterminancy and the formal derivation of programs, *Communications of the ACM* **18**, 1975, 453-457.

[Dou91] Douglis, F., Transparent process migration: Design alternatives and the Sprite implementation, *Software Practice and Experience* **21**, 1991, 757-785.

[EJW96] Elnozahy, E.N., Johnson, D.B., Wang, Y.M., A survey of rollback-recovery protocols in message-passing systems, Technical Report, School of Computer Science, Carnegie Mellon University, 1996.

[Fly72] Flynn, M.J., Some computer organizations and their effectiveness, *IEEE Transactions on Computers* **21**, 1972, 948-960.

[GBD94] Geist, A., Beguelin, A., Dongarra, J., *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, 1994.

[GC93] Guedes, P., Castro, M., Distributed shared object memory, *Proc. Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, Napa, 1993, 142-149.

[GST+93] Gupta, A., Singh, J.P., Truman, J., Hennessy, J.L., An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors, *Proc. Supercomputing'93*, 1993, 214-225.

[Her97] Herdeg, G.A., Design and implementation of the Alpha Server 4100 CPU and memory architecture, *Digital Technical Journal* **8**, 1997, 48-60.

[Hoa74] Hoare, C.A.R., Monitors: An operating system structuring concept, *Communications of the ACM* **17**, 1974, 549-557.

[Div95] Intel Corporation Supercomputer Systems Division, *Paragon System User's Guide*, 1995.

[JS98] Jiang, D., Singh, J.P., A methodology and an evaluation of the SGI Origin2000, *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1998, 171-181.

[Joh88] Johnson, E.E., Completing an MIMD multiprocessor taxonomy, *Computer Architecture News* **16**, 1988, 44-47.

[KDC+94] Keleher, P., Dwarkadas, S., Cox, A.L., Zwaenepoel, W., TreadMarks: Distributed shared memory on standard workstations and operating systems, *Proc. the Winter 94 Usenix Conference*, 1994, 115-131.

[KYA96] Kim, K.H., You, J.H., Abouelnaga, A., A scheme for coordinated independent execution of independently designed recoverable distributed processes, *Proc. IEEE Fault-Tolerant Computing Symposium*, 1996, 130-135.

[Lam78] Lamport, L., Time, clocks and the ordering of events in a distributed system, *Communications of the ACM* **21**, 1978, 558-565.

[LL94] Laudon, J., Lenoski, D., The SGI Origin: A ccNUMA highly scalable server, Technical Report, Silicon Graphics, Inc., 1994.

[Lea93] Lea, R., Cool: System support for distributed programming, *Communications of the ACM* **36**, 1993, 37-46.

[LLT+93] Lenoski, D., Laudon, J., Truman, J., Nakahira, D., Stevens, L., Gupta, A., Hennessy, J., The DASH prototype: Login overhead and performance, *IEEE Transactions on Parallel and Distributed Systems* **4**, 1993, 41-61.

[MPI95] Message Passing Interface Forum, MPI: A message-passing interface standard - version 1.1, 1995.

[MV93] Moos, H., Verbaeten, P., Object migration in a heterogeneous world - a multi-dimensional affair, *Proc. Third International Workshop on Object Orientation in Operating Systems*, Asheville, 1993, 62-72.

[Nut94] Nuttall, M., A brief survey of systems providing process or object migration facilities, *ACM SIGOPS Operating Systems Review* **28**, 1994, 64-80.

[Pie89] Pierce, P., A concurrent file system for a highly parallel mass storage system, *Proc. the Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989, 155-160.

[PFD+89] Pratt, T.W., French, J.C., Dickens, P.M., Janet, S.A., Jr., A comparison of the architecture and performance of two parallel file systems, *Proc. Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989, 161-166.

[RK79] Reed, D.P., Kanodia, R.K., Synchronization with eventcounts and sequencers, *Communications of the ACM* **22**, 1979, 115-123.

[Roy93] Roy, P.J., Unix file access and caching in a multicomputer environment, *Proc. the Usenix Mach III Symposium*, 1993, 21-37.

[SG97] Scales, D.J., Gharachorloo, K., Towards transparent and efficient software distributed shared memory, *Proc. 16th ACM Symposium on Operating Systems Principles (SIGOPS'97), ACM SIGOPS Operating Systems Review* **31**, 1997, 157-169.

[SG94] Silberschatz, A., Galvin, P.B., *Operating System Concepts*, Addison-Wesley, 1994.

[SS94] Singhal, M., Shivaratri, N.G., *Advanced Concepts in Operating Systems*, McGraw Hill, Inc., New York, 1994.

[SH97] Smith, P., Hutchinson, N.C., Heterogeneous process migration: The Tui system, Technical Report, Department of Computer Science, University of British Columbia, 1997.

[TMC90a] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, 1990.

[TMC90b] Thinking Machines Corporation, *Connection Machine Model CM-5 Technical Summary*, 1990.

[Zay87] Zayas, E.R., Attacking the process migration bottleneck, *Proc. Symposium on Operating Systems Principles*, Austin, 1987, 13-22.