```
/* Matrix-vector product Ab=c with parallel linear combination*/
/* Column-oriented distribution of A */
/* Replicated distribution of vectors b and c */
local_m=m/p;
for (i=0; i<n; i++) d[i] = 0;
for (j=0; j<local_m; j++)
    for (i=0 ;i<n; i++)
        d[i] = d[i] + local b[j] * local_A[i][j];
single_accumulation(d,n,c,ADD,1);
single_broadcast(c,1);
```

Fig. 3.13 Program fragment in C notation for a parallel program of the matrix-vector product with column-oriented blockwise distribution of the matrix **A** and reduction operation to compute the result vector **c**. The program uses local array `d` for the parallel computation of partial linear combinations.

`single_accumulation(d,local_m,c,ADD,1)`
denotes an accumulation operation, for which each processor provides its array `d` of size n, and `ADD` denotes the reduction operation. The last parameter is 1 and means that processor P_1 is the root processor of the operation, which stores the result of the addition into the array `c` of length n. The final `single_broadcast(c,1)` sends the array `c` from processor P_1 to all other processors and a replicated distribution of `c` results.

Alternatively to this final communication, multi-accumulation operation can be applied which leads to a block-wise distribution of array `c`. This program version may be advantageous if `c` is required to have the same distribution as array `b`. Each processor accumulates the n/p elements of the local arrays `d`, i.e., each processor computes a block of the result vector `c` and stores it in its local memory. This communication is illustrated in Fig. 3.14(2b).

For shared memory machines, the parallel computation of the linear combinations can also be used but special care is needed to avoid access conflicts for the write accesses when computing the partial linear combinations. To avoid write conflicts, a separate array `d_k` of length n should be allocated for each of the processors P_k to compute the partial result in parallel without conflicts. The final accumulation needs no communication, since the data `d_k` are in the common memory, and can be performed in a blocked way.

The computation and communication time for the matrix-vector product is analyzed in Sect. 4.4.2.

3.8 Processes and Threads

Parallel programming models are often based on processors or threads. Both are abstractions for a flow of control, but there are some differences which we will con-

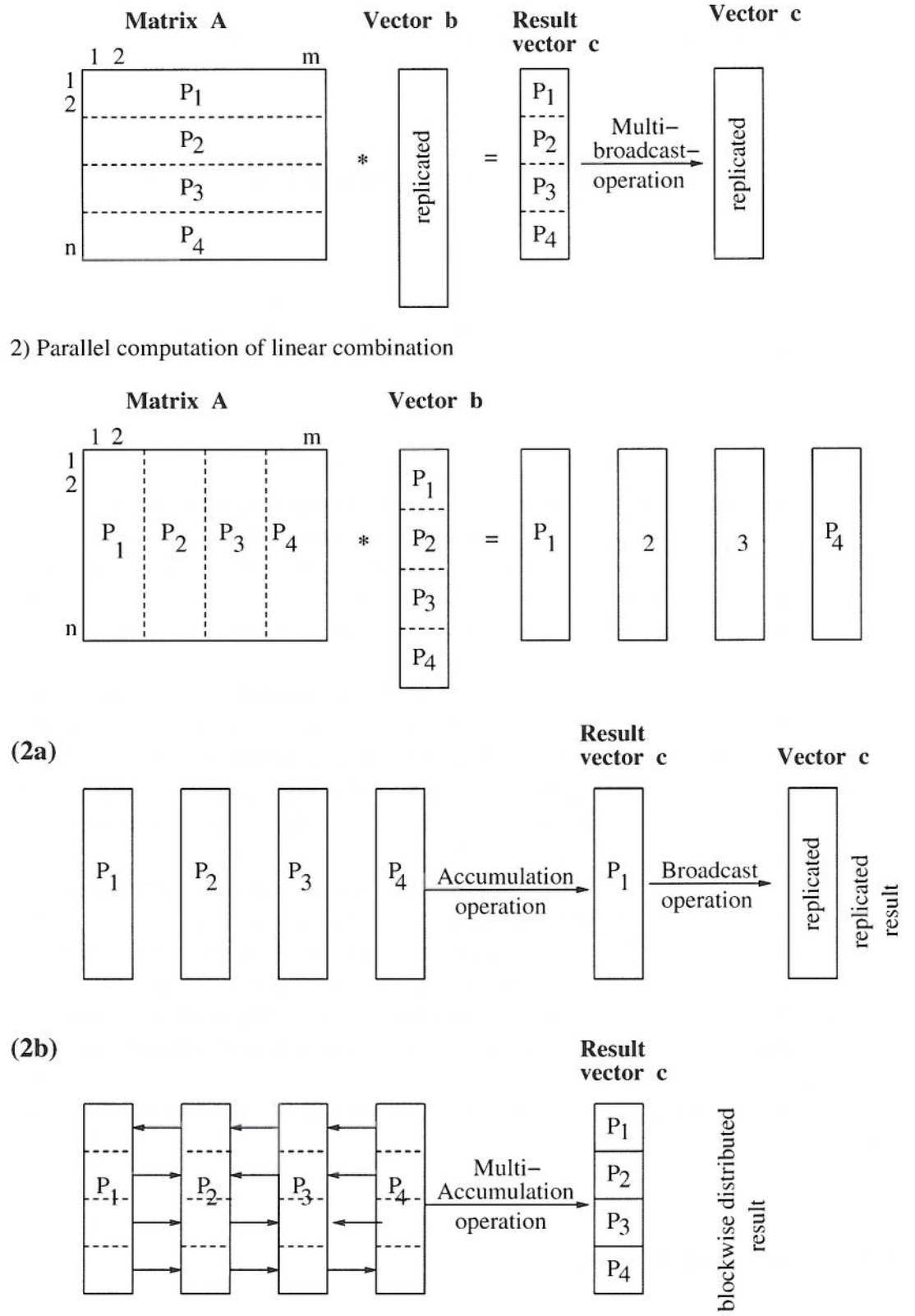

Fig. 3.14 Parallel matrix-vector multiplication with 1) parallel computation of scalar products and replicated result and 2) parallel computation of linear combinations with **(a)** replicated result and **(b)** blockwise distribution of the result.

sider in this section in more detail. As described in Sect. 3.2, the principal idea is to decompose the computation of an application into tasks and to employ multiple control flows running on different processors or cores for their execution, thus obtaining a smaller overall execution time by parallel processing.

3.8.1 Processes

In general, a process is defined as a program in execution. The process comprises the executable program along with all information that is necessary for the execution of the program. This includes the program data on the runtime stack or the heap, the current values of the registers, as well as the content of the program counter which specifies the next instruction to be executed. All this information changes dynamically during the execution of the process. Each process has its own address space, i.e., the process has exclusive access to its data. When two processes want to exchange data, this has to be done by explicit communication.

A process is assigned to execution resources (processors or cores) for execution. There may be more processes than execution resources. To bring all processes to execution from time to time, an execution resource typically executes several processes at different points in time, e.g. in a round-robin fashion. If the execution is assigned to another process by the scheduler of the operating system, the state of the suspended process must be saved to allow a continuation of the execution at a later time with the process state before suspension. This switching between processes is called **context switch**, and it may cause a significant overhead, depending on the hardware support [155]. Often time slicing is used to switch between the processes. If there is a single execution resource only, the active processes are executed concurrently in a time-sliced way, but there is no real parallelism. If several execution resources are available, different processes can be executed by different execution resources, thus indeed leading to a parallel execution.

When a process is generated, it must obtain the data required for its execution. In Unix systems, a process P_1 can create a new process P_2 with the `fork` system call. The new child process P_2 is an identical copy of the parent process P_1 at the time of the `fork` call. This means, that the child process P_2 works on a *copy* of the address space of the parent process P_1 and executes the same program as P_1, starting with the instruction following the `fork` call. The child process gets its own process number and, depending on this process number, it can execute different statements as the parent process. Since each process has its own address space and since process creation includes the generation of a copy of the address space of the parent process, process creation and management may be quite time-consuming. Data exchange between processes is often done via socket communication which is based on TCP/IP or UDP/IP communication. This may lead to a significant overhead, depending on the socket implementation and the speed of the interconnection between the execution resources assigned to the communicating processes.

3.8.2 Threads

The thread model is an extension of the process model. In the thread model, each process may consist of *multiple* independent control flows which are called **threads**. The word *thread* is used to indicate that a potentially long continuous sequence of instructions is executed. During the execution of a process, the different threads of this process are assigned to execution resources by a scheduling method.

3.8.2.1 Basic concepts of threads

A significant features of threads is that the threads of *one* process share the address space of the process, i.e., they have a common address space. When a thread stores a value in the shared address space, another thread of the same process can access this value afterwards. Threads are typically used if the execution resources used have access to a physically shared memory, as is the case for the cores of a multicore processor. In this case, information exchange is fast compared to socket communication. Thread generation is usually much faster than process generation: no copy of the address space is necessary since the threads of a process share the address space. Therefore, the use of threads is often more flexible than the use of processes, yet providing the same advantages concerning a parallel execution. In particular, the different threads of a process can be assigned to different cores of a multicore processor, thus providing parallelism within the processes.

Threads can be provided by the runtime system as **user-level threads** or by the operating system as **kernel threads**. User-level threads are managed by a *thread library* without specific support of the operating system. This has the advantage that a switch from one thread to another can be done without interaction of the operating system and is therefore quite fast. Disadvantages of the management of threads at user-level come from the fact that the operating system has no knowledge about the existence of threads and manages entire processes only. Therefore, the operating system cannot map different threads of the same process to different execution resources and all threads of one process are executed on the same execution resource. Moreover, the operating system cannot switch to another thread if one thread executes a blocking I/O operation. Instead, the CPU scheduler of the operating system suspends the entire process and assigns the execution resource to another process.

These disadvantages can be avoided by using kernel threads, since the operating system is aware of the existence of threads and can react correspondingly. This is especially important for an efficient use of the cores of a multicore system. Most operating systems support threads at the kernel level.

3.8.2.2 Execution models for threads

If there is no support for thread management by the operating system, the thread library is responsible for the entire thread scheduling. In this case, *all* user-level threads of a user process are mapped to *one* process of the operating system. This is called **N:1 mapping**, or *many-to-one mapping*, see Fig. 3.15 for an illustration. At each point in time, the library scheduler determines which of the different threads comes to execution. The mapping of the processes to the execution resources is done by the operating system. If several execution resources are available, the operating system can bring several processes to execution concurrently, thus exploiting parallelism. But with this organization, the execution of different threads of one process on different execution resources is not possible.

If the operating system supports thread management, there are two possibilities for the mapping of user-level threads to kernel threads. The first possibility is to generate a kernel thread for each user-level thread. This is called **1:1 mapping**, or *one-to-one mapping*, see Fig. 3.16 for an illustration. The scheduler of the operating system selects which kernel threads are executed at which point in time. If multiple execution resources are available, it also determines the mapping of the kernel threads to the execution resources. Since each user-level thread is assigned to exactly one kernel thread, there is no need for a library scheduler. Using a 1:1 mapping, different threads of a user process can be mapped to different execution resources, if enough resources are available, thus leading to a parallel execution within a single process.

The second possibility is to use a two-level scheduling where the scheduler of the thread library assigns the user-level threads to a given set of kernel threads. The scheduler of the operating system maps the kernel threads to the available execution resources. This is called **N:M mapping**, or *many-to-many mapping*, see Fig. 3.17 for an illustration. At different points in time, a user thread may be mapped to a different kernel thread, i.e., no fixed mapping is used. Correspondingly, at different points in

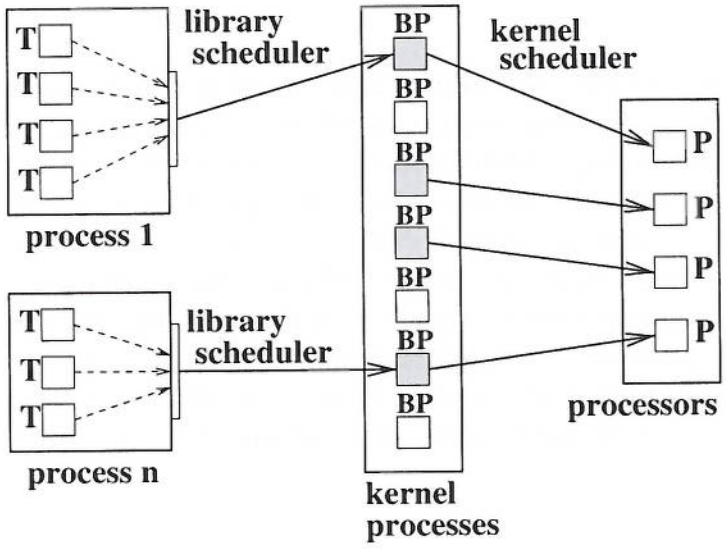

Fig. 3.15 Illustration of a N:1 mapping for thread management without kernel threads. The scheduler of the thread library selects the next thread T of the user process for execution. Each user process is assigned to exactly one process BP of the operating system. The scheduler of the operating system selects the processes to be executed at a certain time and maps them to the execution resources P.

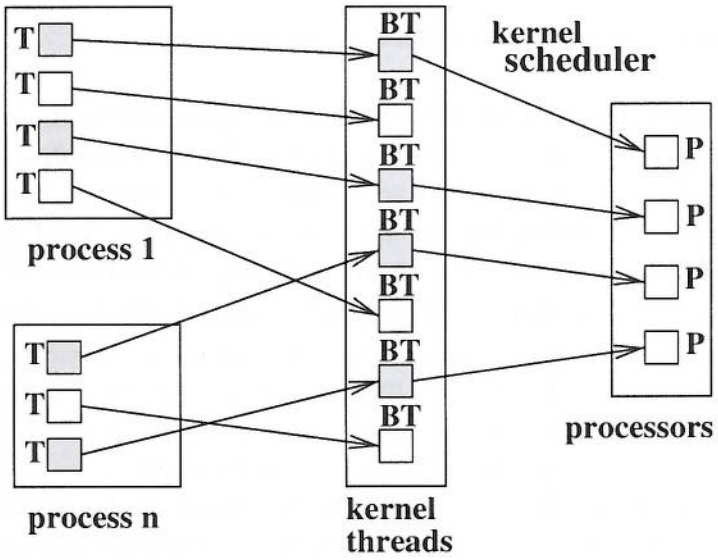

Fig. 3.16 Illustration of a 1:1 mapping for thread management with kernel threads. Each user-level thread T is assigned to one kernel thread BT. The kernel threads BT are mapped to execution resources P by the scheduler of the operating system.

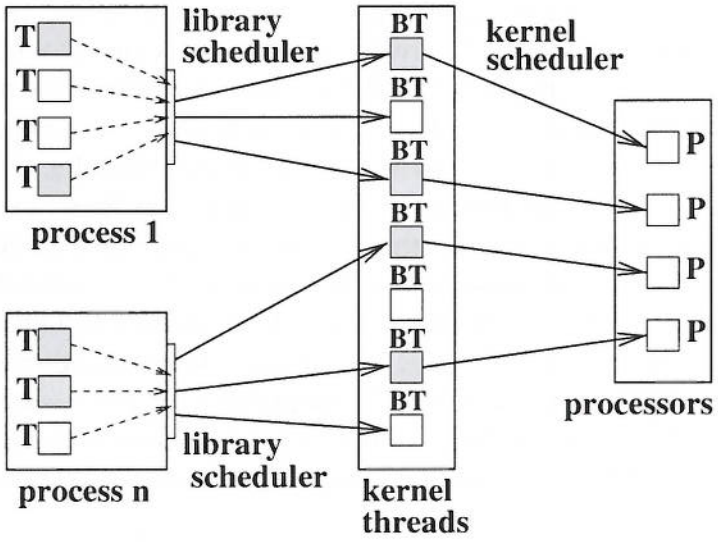

Fig. 3.17 Illustration of a N:M mapping for thread management with kernel threads using a two-level scheduling. User-level threads T of different processes are assigned to a set of kernel threads BT (N:M mapping) which are then mapped by the scheduler of the operating system to execution resources P.

time, a kernel thread may execute different user threads. Depending on the thread library, the programmer can influence the scheduler of the library, e.g., by selecting a scheduling method as this is the case for the Pthreads library, see Sect. 6.1.10 for more details. The scheduler of the operating system on the other hand is tuned for an efficient use of the hardware resources, and there is typically no possibility for the programmer to directly influence the behavior of this scheduler. This second mapping possibility usually provides more flexibility than a 1:1 mapping, since the programmer can adapt the number of user-level threads to the specific algorithm or application. The operating system can select the number of kernel threads such that an efficient management and mapping of the execution resources is facilitated.

3.8.2.3 Thread states

A thread can be in one of the following states:

- **newly generated**, i.e., the thread has just been generated, but has not yet performed any operation;
- **executable**, i.e., the thread is ready for execution, but is currently not assigned to any execution resources;
- **running**, i.e., the thread is currently being executed by an execution resource;
- **waiting**, i.e., the thread is waiting for an external event to occur; the thread cannot be executed before the external event happens;
- **finished**, i.e., the thread has terminated all its operations.

Fig. 3.18 illustrates the transition between these states. The transitions between the states *executable* and *running* are determined by the scheduler. A thread may enter the state *waiting* because of a blocking I/O operation or because of the execution of a synchronization operation which causes it to be blocked. The transition from state *waiting* to *executable* may be caused by a termination of a previously issued I/O operation or because another thread releases the resource which this thread is waiting for.

3.8.2.4 Visibility of data

The different threads of a process share a common address space. This means that the global variables of a program and all dynamically allocated data objects can be accessed by any thread of this process, no matter which of the threads has allocated the object. But for each thread, there is a private runtime stack for controlling function calls of this thread and to store the local variables of these functions, see Fig. 3.19 for an illustration. The data kept on the runtime stack is local data of the corresponding thread and the other threads have no direct access to this data. It is in principle possible to give them access by passing an address, but this is dangerous, since how long the data remains accessible cannot be predicted. The stack frame of a function call is freed as soon as the function call is terminated. The runtime stack of a thread exists only as long as the thread is *active*; it is freed as soon as the thread is terminated. Therefore, a return value of a thread should not be passed via its runtime stack. Instead, a global variable or a dynamically allocated data object should be used, see Chap. 6 for more details.

3.8.3 Synchronization mechanisms

When multiple threads execute a parallel program in parallel, their execution has to be coordinated to avoid race conditions. Synchronization mechanisms are provided to enable a coordination, e.g., to ensure a certain execution order of the threads or to control access to shared data structures. Synchronization for shared variables

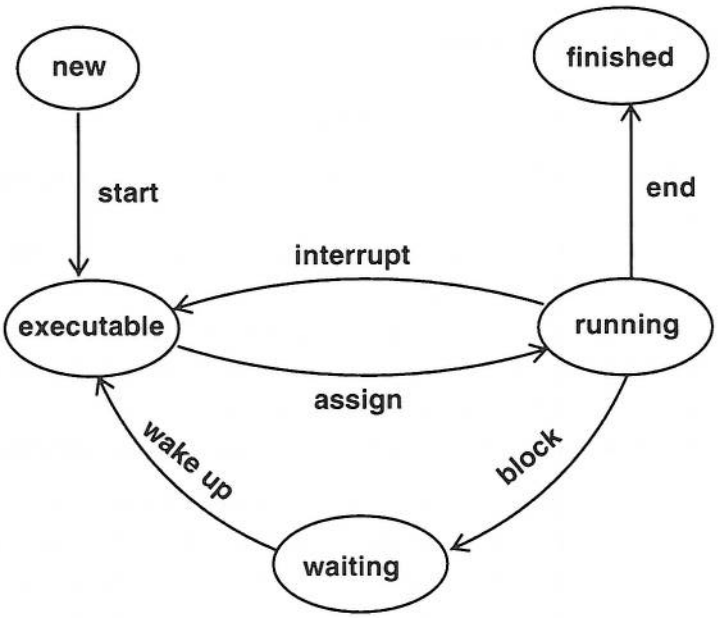

Fig. 3.18 States of a thread. The nodes of the diagram show the possible states of a thread and the arrows show possible transitions between them.

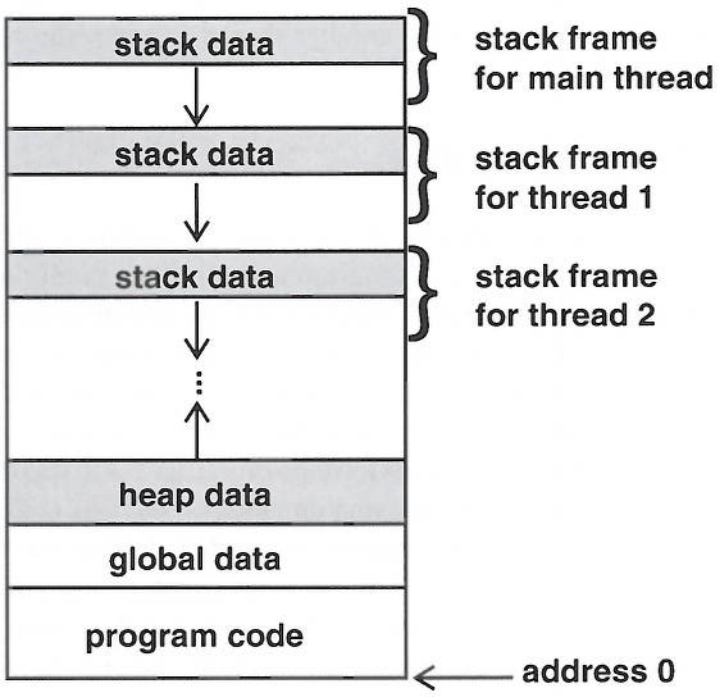

Fig. 3.19 Runtime stack for the management of a program with multiple threads.

is mainly used to avoid a concurrent manipulation of the same variable by different threads, which may lead to non-deterministic behavior. This is important for multi-threaded programs, no matter whether a single execution resource is used in a time-slicing way, or whether several execution resources execute multiple threads in parallel. Different synchronization mechanisms are provided for different situations. In the following, we give a short overview.

3.8.3.1 Lock synchronization

For a concurrent access of shared variables, race conditions can be avoided by a **lock mechanism** based on predefined **lock variables**, which are also called **mutex variables** as they help to ensure mutual exclusion. A lock variable `l` can be in one of two states: *locked* or *unlocked*. Two operations are provided to influence this state: `lock(l)` and `unlock(l)`. The execution of `lock(l)` locks `l` such that it cannot be locked by another thread; after the execution, `l` is in the *locked* state and the thread that has executed `lock(l)` is the *owner* of `l`. The execution of `unlock(l)` unlocks a previously locked lock variable `l`; after the execution, `l` is in the *unlocked* state and has no owner. To avoid race conditions for the execution of a program part, a lock variable `l` is assigned to this program part and each thread executes `lock(l)` before entering the program part and `unlock(l)` after leaving the program part. To avoid race conditions, *each* of the threads must obey this programming rule.

A call of `lock(l)` for a lock variable `l` has the effect that the executing thread T_1 becomes the owner of `l`, if `l` has been in the *unlocked* state before. But if there is already another owner T_2 of `l` before T_1 calls `lock(l)`, T_1 is blocked until T_2 has called `unlock(l)` to release `l`. If there are blocked threads waiting for `l` when `unlock(l)` is called, one of the waiting threads is woken up and becomes the new owner of `l`. Thus, using a lock mechanism in the described way leads to a *sequentialization* of the execution of a program part which ensures that at each point in time, only one thread executes the program part. The provision of lock mechanisms in libraries like Pthreads, OpenMP, or Java threads is described in Chap. 6.

It is important to see that mutual exclusion for accessing a shared variable can only be guaranteed if all threads use a lock synchronization to access the shared variable. If this is not the case, a race condition may occur, leading to an incorrect program behavior. This can be illustrated by the following example where two threads T_1 and T_2 access a shared integer variable s which is protected by a lock variable `l` [127]

Thread T_1	Thread T_2
`lock(l);`	
`s = 1;`	`s = 2;`
`if(s! = 1) fire_missile();`	
`unlock(l);`	

In this example, thread T_1 may get interrupted by the scheduler and thread T_2 can set the value of `s` to 2; if T_1 resumes execution, `s` has value 2 and `fire_missile()` is called. For other execution orders, `fire_missile()` will not be called. This nondeterministic behavior can be avoided if T_2 also uses a lock mechanism with `l` to access `s`.

Another mechanism to ensure mutual exclusion is provided by **semaphores** [46]. A semaphore is a data structure which contains an integer counter s and to which two atomic operations $P(s)$ and $V(s)$ can be applied. A *binary semaphore* s can only have values 0 or 1. For a *counting semaphore*, s can have any positive integer value. The operation $P(s)$, also denoted as `wait(s)`, waits until the value of s is

larger than 0. When this is the case, the value of s is decreased by 1, and execution can continue with the subsequent instructions. The operation $V(s)$, also denoted as `signal(s)`, increments the value of s by 1. To ensure mutual exclusion for a critical section, the section is protected by a semaphore s in the following form:

```
wait(s)
critical section
signal(s)
```

Different threads may execute operations $P(s)$ or $V(s)$ for a semaphore s to access the critical section. After a thread T_1 has successfully executed the operation `wait(s)` with waiting it can enter the critical section. Every other thread T_2 is blocked when it executes `wait(s)` and can therefore not enter the critical section. When T_1 executes `signal(s)` after leaving the critical section, one of the waiting threads will be woken up and can enter the critical section.

Another concept to ensure mutual exclusion is the concept of **monitors** [100]. A monitor is a language construct which allows the definition of data structures and access operations. These operations are the *only* means with which the data of a monitor can be accessed. The monitor ensures that the access operations are executed with mutual exclusion, i.e., at each point in time, only one thread is allowed to execute any of the access methods provided.

3.8.3.2 Thread execution control

To control the execution of multiple threads, barrier synchronization and condition synchronization can be used. A **barrier synchronization** defines a synchronization point where each thread must wait until all other threads have also reached this synchronization point. Thus, none of the threads executes any statement after the synchronization point before all other threads have also arrived at this point. A barrier synchronization also has the effect that it defines a global state of the shared address space in which all operations specified before the synchronization point have been executed. Statements after the synchronization point can be sure that this global state has been established.

Using a **condition synchronization**, a thread T_1 is blocked until a given condition has been established. The condition could, for example, be that a shared variable contain a specific value or have a specific state like a shared buffer containing at least one entry. The blocked thread T_1 can only be woken up by another thread T_2, e.g., after T_2 has established the condition which T_1 waits for. When T_1 is woken up, it enters the state *executable*, see Sect. 3.8.2.2, and will later be assigned to an execution resource, then entering the state *running*. Thus, after being woken up, T_1 may not be immediately executed, e.g., if not enough execution resources are available. Therefore, although T_2 may have established the condition which T_1 waits for, it is important that T_1 check the condition again as soon as it is running. The reason for this additional check is that in the meantime another thread T_3 may have performed some computations which might have led to the fact that the condition

is not fulfilled any more. Condition synchronization can be supported by condition variables. These are for example provided by Pthreads and must be used together with a lock variable to avoid race condition when evaluating the condition, see Sect. 6.1 for more details. A similar mechanism is provided in Java by `wait()` and `notify()`, see Sect. 6.2.3.

3.8.4 Developing efficient and correct thread programs

Depending on the requirements of an application and the specific implementation by the programmer, synchronization leads to a complicated interaction between the executing threads. This may cause problems like performance degradation by sequentializations, or even deadlocks. This section contains a short discussion of this topic and gives some suggestions how efficient thread-based programs can be developed.

3.8.4.1 Number of threads and sequentialization

Depending on the design and implementation, the runtime of a parallel program based on threads can be quite different. For the design of a parallel program it is important

- to use a suitable number of threads which should be selected according to the degree of parallelism provided by the application and the number of execution resources available and
- to avoid sequentialization by synchronization operations whenever possible.

When synchronization is necessary, e.g., to avoid race conditions, it is important that the resulting critical section which is executed sequentially, is made as small as possible to reduce the resulting waiting times.

The creation of threads is necessary to exploit parallel execution. A parallel program should create a sufficiently large number of threads to provide enough work for all cores of an execution platform, thus using the available resources efficiently. But the number of threads created should not be too large to keep the overhead for thread creation, management and termination small. For a large number of threads, the work per thread may become quite small, giving the thread overhead a signification portion of the overall execution time. Moreover, many hardware resources, in particular caches, may be shared by the cores, and performance degradations may result, if too many threads share the resources; in the case of caches, a degradation of the read/write bandwidth might result.

The threads of a parallel program must be coordinated to ensure a correct behavior. An example is the use of synchronization operations to avoid race conditions. But too many synchronizations may lead to situations where only one or a small number of threads are active while the other threads are waiting because of a synchronization operation. In effect, this may result in a **sequentialization** of the thread execution, and

the available parallelism cannot be used. In such situations, increasing the number of threads does not lead to faster program execution, since the new threads are waiting most of the time.

3.8.4.2 Deadlock

Non-deterministic behavior and race conditions can be avoided by synchronization mechanisms like lock synchronization. But the use of locks can lead to **deadlocks**, when program execution comes into a state where each thread waits for an event that can only be caused by another thread, but this thread is also waiting. Generally, a deadlock occurs for a set of activities, if each of the activities waits for an event that can only be caused by one of the other activities, such that a cycle of mutual waiting occurs. A deadlock may occur in the following example where two threads T_1 and T_2 both use two locks `s1` and `s2`:

Thread T_1	Thread T_2
`lock(s1);`	`lock(s2);`
`lock(s2);`	`lock(s1) :`
`do_work();`	`do_work();`
`unlock(s2)`	`unlock(s1)`
`unlock(s1)`	`unlock(s2)`

A deadlock occurs for the following execution order:

- a thread T_1 first tries to set a lock s_1, and then s_2; after having locked s_1 successfully, T_1 is interrupted by the scheduler;
- a thread T_2 first tries to set lock s_2 and then s_1; after having locked s_2 successfully, T_2 waits for the release of s_1.

In this situation, s_1 is locked by T_1 and s_2 by T_2. Both threads T_1 and T_2 wait for the release of the missing lock by the other thread. But this cannot occur, since the other thread is waiting.

It is important to avoid such mutual or cyclic waiting situations, since the program cannot be terminated in such situations. Specific techniques are available to avoid deadlocks in cases where a thread must set multiple locks to proceed. Such techniques are described in Sect. 6.1.2.

3.8.4.3 Memory access times and cache effects

Memory access times may constitute a significant portion of the execution time of a parallel program. A memory access issued by a program causes a data transfer from the main memory into the cache hierarchy of that core which has issued the memory

access. This data transfer is caused by the read and write operations of the cores. Depending on the specific pattern of read and write operations, there is not only a transfer from main memory to the local caches of the cores, but there may also be a transfer between the local caches of the cores. The exact behavior is controlled by hardware, and the programmer has no direct influence on this behavior.

The transfer within the memory hierarchy can be captured by dependencies between the memory accesses issued by different cores. These dependencies can be categorized as read-read dependency, read-write dependency and write-write dependency. A read-read dependency occurs if two threads running on different cores access the same memory location. If this memory location is stored in the local caches of both cores, both can read the stored values from their cache, and no access to main memory needs to be done. A read-write dependency occurs, if one thread T_1 executes a write into a memory location which is later read by another thread T_2 running on a different core. If the two cores involved do not share a common cache, the memory location that is written by T_1 must be transferred into main memory after the write before T_2 executes its read which then causes a transfer from main memory into the local cache of the core executing T_2. Thus, a read-write dependency consumes memory bandwidth.

A write-write dependency occurs, if two threads T_1 and T_2, running on different cores perform a write into the same memory location in a given order. Assuming that T_1 writes before T_2, a cache coherency protocol, see Sect. 2.7.3, must ensure that the caches of the participating cores are notified when the memory accesses occur. The exact behavior depends on the protocol and the cache implementation as write-through or write-back, see Sect. 2.7.1. In any case, the protocol causes a certain amount of overhead to handle the write-write dependency.

False sharing occurs if two threads T_1 and T_2, running on different cores, access different memory locations that are held in the same cache line. In this case, the same memory operations must be performed as for an access to the same memory locations, since a cache line is the smallest transfer unit in the memory hierarchy. False sharing can lead to a significant amount of memory transfers and to notable performance degradations. It can be avoided by an alignment of variables to cache line boundaries; this is supported by some compilers.

3.9 Further parallel programming approaches

For the programming of parallel architectures, a large number of approaches have been developed during the last years. A first classification of these approaches can be made according to the memory view provided, shared address space or distributed address space, as discussed earlier. In the following, we give a detailed description of the most popular approaches for both classes. For a distributed address space, MPI is by far the most often used environment, see Chap. 5 for a detailed description. The use of MPI is not restricted to parallel machines with a physically distributed memory organization. It can also be used for parallel architectures with a physically