Parallel computing

The execution time of a program can be shortened by distributing various tasks to different processors. This involves its own problems, like synchronization of the processes. In this course we cannot go to the details of developing parallel programs, since considerable amount of programming experience is needed. (In the worst case making a program parallel may even slow it down considerably!) Therefore, we'll just mention some basic concepts.

As a matter of fact, many of them are already needed in a multitasking environment on a single processors, where several processes are given small time slices. Therefore, in the following we'll not make a strong difference between a process and processor.

In algorithmic languages the order of execution is exactly specified. However, often the exact order is not important.

```
do i=1,100
  a(i) = i
end do
```

The cycles of this loop are independent; thus they could be executed in any order.

Independent statements can be distributed to be executed on different processors.

SISD = Single Instruction Single Data, the traditional architecture.

SIMD = Single Instruction Multiple Data, the same code is executed for different pieces of data (like for different elements of an array on a vector processor)

MIMD = Multiple Instruction Multiple Data, different processors can execute different things for different data

SMP = Symmetric MultiProcessor, different processors share a common memory

MMP = Massively Parallel Processor, each processor has its own memory

Problems

Programming is much more complicated than in the case of a single processor:

– Order of execution of the parallel parts may vary in a random manner. When needed, the execution must be synchronized.

– It is impossible to test all possible orders of execution. How to guarantee that no order will lead to an error?

– It must be guaranteed that several processors will not try to modify the same variable at the same time.

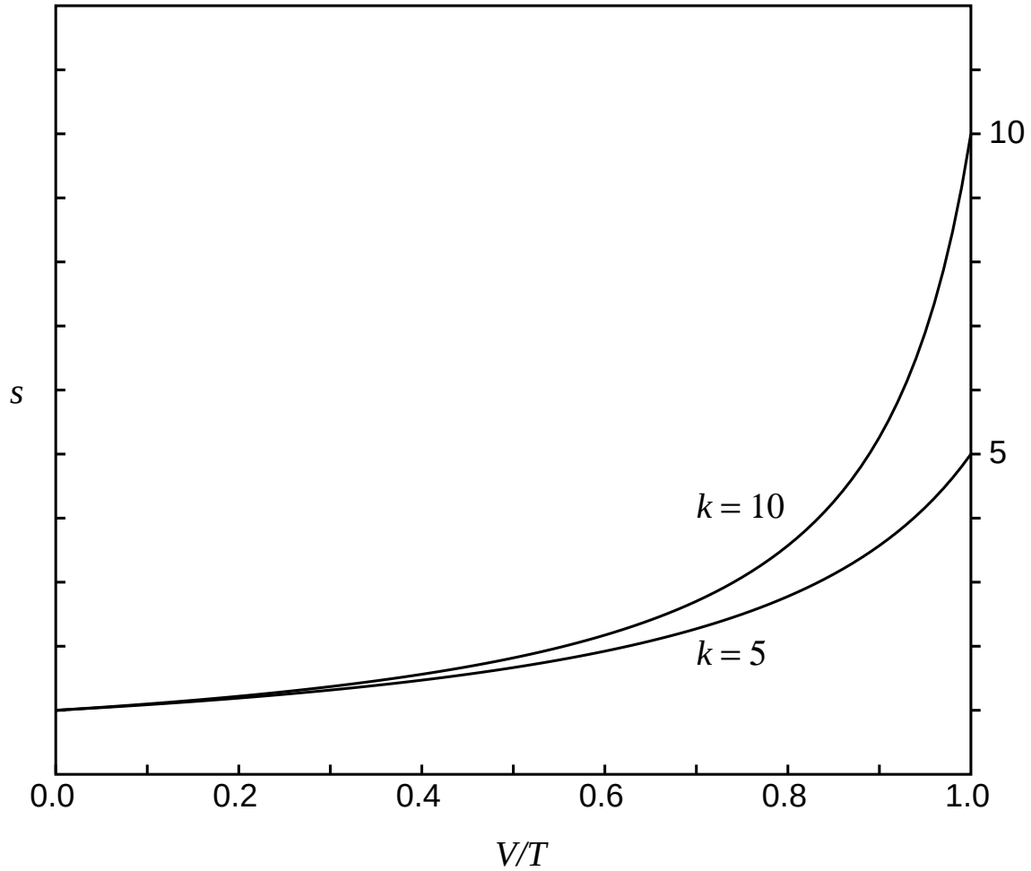
– Different processors may have to wait for each others, and the program cannot continue (deadlock).

Amdahl's law: the speedup of execution is

$$s = \frac{1}{1 - v + \frac{v}{k}},$$

where k is the number of processors and v the fraction of the parallel code.

The part of the code that cannot be vectorized / parallelized has a strong influence on the execution time.

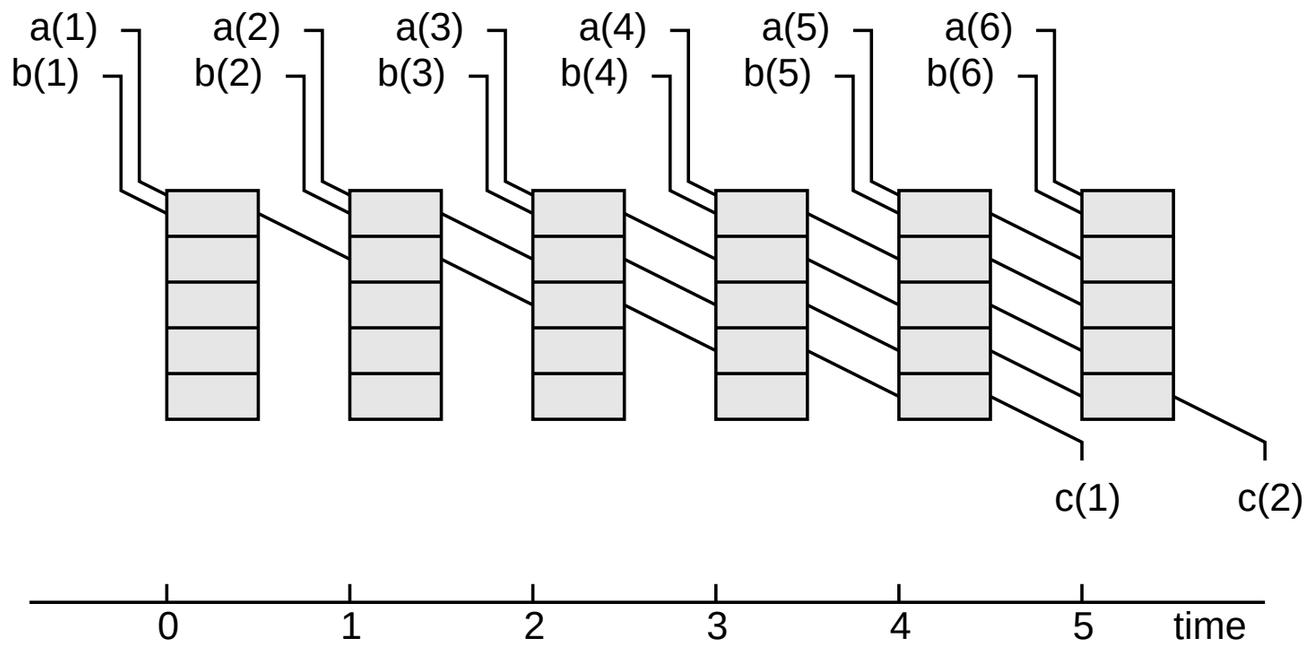


A vector processor is not a multiprocessor machine.

Each operation can be divided into phases. Different phases can executed at the same time for different numbers.

Evaluating the first value takes the same time as with an ordinary single operation, but after that a new ersult is obtained at each clock cycle (pipelining).

Example: an array operation `c = a+b`:



Protecting variables

Critical region is a part of a program that can be executed by one process only at a time.

For example, several processes want to update the same data structure. Simultneous updates must be prevented.

A lock is an integer (0=open, 1=closed):

```
LOCK(s)
critical region

UNLOCK(s)
```

LOCK will wait till the variable s is zero 0. When that happens, it will change s to 1 and proceed. Testing and setting a lock variable must be executed as an operation that cannot be interrupted. Usually machine languages have commands for this.

Active waiting: a process keeps testing the lock continuously until it can proceed.

A critical region can also be protected by a ısemaphore. A semaphore is a data structure containing an integer and a queue for processes waiting for execution.

Dijkstra's P and V operations can be used for handling semaphores:

```
P(sem):
  if (sem == 1) sem = 0
  else
   stay in the queue and wait

V(sem):
  if (the queue contains processes waiting for the semaphore)
    start the next process in the queue
  else
   sem = 1
```

P and V must be operations that cannot be interrupted.

A process waiting in the queue does not use processor time. When the critical region is free, the process is moved to the ready queue and is ready to continue execution as soon as the processor becomes available.

Synchronization

In multiprocessor architectures independent parts of a program can be executed at the same time on different processors.

```
a = sin(x)
b = sqrt(x)
c = a + b
```

The two first statements can be executed simultaneously, but the third one can be executed only after the execution of the two previous ones is finished. Otherwise the result would be random depending on whwther the values of a and be have already been changed.

We need ısynchronization to guarantee that the various branches of a parallel program are finished before proceeding to a part of the program that depends on the results of the parallel part.

Processes can communicate by sending messages.

```
program 1:

  produce data
  signal(ready)
  ...


program 2:

  ...
  wait(redy)
  consume data
  ...
```

Program 2 will wait until it receives the required message.

Dependences

Dependences between statements restrict parallelization.

```
do i=2,100
  a(i) = a(i-1)+1
end do
```

Each element of the array depends on the value obtained on the previous cycle. This is a *recursion* that prevents parallelization (and vectorization).

The next loop can be vectorized , since the loop does not depend on the previously calculated values:

```
do i=1,100
  a(i) = a(i+1)+1
end do
```

Yet the loop cannot be parallelized (without some additional conditions).

The following loop has no dependencies:

```
do i=1,100
  a(i) = a(i+100)+1
end do
```

Compilers can detect many dependences automatically. If a dependence is possible, the loop is not vectorized / parallellized.

In the following the dependence depends on the value of the variable n, and cannot be determined automatically:

```
do i=1,100
  a(i) = a(i+n)+1
end do
```

A directive can be used to tell the compiler that it is safe to vectorize / parallelize the loop.

Directives are just comments with a certain syntax; thus they do not interfere with compilation in other environments.

Many hardware manufacturers have their own compilers undrstanding their own directives. Check them in the manual of the compiler.

HPF

High Performance Fortran is an extension of Fortran for data parallel computation (SIMD). Also, since F95 the lnguage itself has some features for parallel programming (like the forall statament).

It is not possible to execute arbitrary parts of a program in parallel.

Directives are used to direct execution:

```
real, dimension(1000):: a
!HPF$ DISTRIBUTE a (BLOCK)
do i=1,1000
  a(i)=sqrt(i/1000.0)
end do
```

This instructs the compiler to divide the array a into similar blocks to different processors.

The size of the block can also be given:

```
!HPF$ DISTRIBUTE (BLOCK(200)) :: a
```

The elements of the array can also be distributed cyclically, in which case successive elements go to different processors:

```
!HPF$ DISTRIBUTE a (CYCLIC)
```

Number of processors can be specified with the directive `PROCESSORS`:

```
real, dimension(1000) :: a
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a
```

This will use 10 processors, the first of which will get the elements a(1:100) etc.

Directives can also be used to define how the processors are connected to each others (topology).

Directive ALIGN tells that different arrays are distributed the same way among processors:

```
real, dimension(1000) :: a, b, c, d
!HPF$ DISTRIBUTE a(BLOCK)
!HPF$ ALIGN WITH a :: b, c
a = b+c

!HPF$ ALIGN WITH a(1) :: d(10)
d(10:1000) = a(1:991)
```

Arrays should be aligned so that each processor will need only (or mainly) its own pieces of them, which will eliminate (or minimize) the need to transfer data between processors.

If a loop contains a function call, it may have side effects that prevent parallelization:

```
do i=1,100
  a(i) = funk(b(i))
end do
```

A function can be given attribute `pure`, meaning it cannot have side effects, and the loop can be parallelized. The attribute is availabl also in F95:

```
pure function funk(x)
real, intent(in) :: x
real :: funk
funk = sqrt(sin(x))
end function
```

The directive `independent` can be used to tell that the iterations of a loop are independent:

```
!HPF$ INDEPENDENT
do i=1,100
  a(i) = a(i+n)+1
end do
```

MPI

Message-Passing Interface is a standardized library for parallel programming.

Programs running on different processors communicate by sending messages to each others. Messages can be used to transfer data and synchronize programs.

The following example is from the MPI guide published by CSC. Copies of the same program are started on several processors.

```
PROGRAM example
 IMPLICIT NONE
 INCLUDE 'mpif.h'
 INTEGER, PARAMETER :: tag = 50
 INTEGER :: id, ntasks, source_id, &
    dest_id, rc, i
 INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
 INTEGER, DIMENSION(2) :: msg
 CALL MPI_INIT(rc)
 IF (rc /= MPI_SUCCESS) THEN
  WRITE(*,*) 'MPI initialization failed'
  STOP
 END IF
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
 IF (id /= 0) THEN
  msg(1) = id
  msg(2) = ntasks
  dest_id = 0
  CALL MPI_SEND(msg, 2, MPI_INTEGER, dest_id, &
     tag, MPI_COMM_WORLD, rc)
```

```
 ELSE
  DO i = 1, ntasks-1
    CALL MPI_RECV(msg, 2, MPI_INTEGER, &
       MPI_ANY_SOURCE, tag, &
       MPI_COMM_WORLD, status, rc)
    source_id = status(MPI_SOURCE)
    WRITE(*,*) 'message:', msg, 'sender:', source_id
  END DO
 END IF
 CALL MPI_FINALIZE(rc)
END PROGRAM
```

In all calls the last parameter (in this example rc) is a return code, which will tell whether the operation was succesful. Its value is `MPI_SUCCESS`, if everything is ok.

`MPI_INIT` initializes a parallel job.

`MPI_FINALIZE` terminates a parallel job.

`CALL MPI_COMM_SIZE` tells how many processors the job will use.

`CALL MPI_COMM_RANK` tells the number of the calling process.

`MPI_SEND` sends a message to another process.

`MPI_RECV` waits till it receives the required message from another process.

SEND and RECV are blocking: execution will continue after the subroutine only after the whole message has been transferred.

There are also nonblocking versions that allow the execution to proceed immediately. That is faster, but the synchronization must be handled explicitly.

Messages between two processes will always arrive in the same order as they were sent. If a message goes through several processes, the order may not be retained.

`MPI_REDUCE` collects similar data from all processes and executes some operation on them. For example, processors can compute pieces of a series expansion, which are then colleced and added.

`MPI_BCAST` (broadcast) sends the same information to all processes.

`MPI_SCATTER` sends different data to different processes. A typical example is distributing an array to several proceses.

`MPI_GATHER` inverse of the previous operation; For example, collect pieces of a table from different processes.

`MPI_BARRIER` synchronization; processes will wait until all of them have reached this point.