Supercomputing - Parallelism

- Why parallel computing?

- Evolution of parallel computing

- Parallelism in Physics

- Concepts of parallel computing: SIMD - MIMD

- Data parallel concepts (HPF)

- Message passing concepts (MPI)

- Simple examples

Why parallel computing?

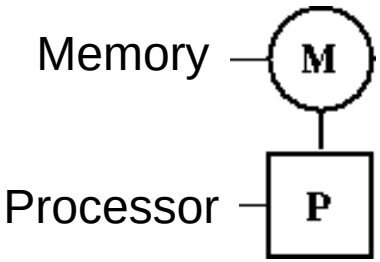
- Simulation of the real world requires enormous computational resources
- Serial processors are obsolete after some time
- Upgrading of parallel hardware by adding more processors (in theory)
- Instead of specially developed single processors many off-the-shelf (cheap) processors in parallel hardware (e.g. linux clusters)
- Physical limit to processor speed (speed of light)

Economics - Physics

Evolution of Parallel Computing

"When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes."

L.F. Menabrea (1842)



Early 1950's: Basic design for electronic computers: a single processing unit connected to a single memory store. One event happening at a time.

John von Neumann

The first parallel computers were built in the 1970's (e.g. ILLIAC IV, Illinois University). The first widely used parallel computers appeared in the 1980's (e.g. ICL, CosmicCube, CM-2 by Thinking Machines Corp.)

Parallelism in Physics

The fundamental laws of physics (e.g. the conservation of energy) are ***parallel*** in nature. They apply all the time at each point (or small volume) in space. The dynamic behaviour of physical phenomena is usually expressed by partial differential equations, e.g. the 1-D wave equation

$$\partial_t^2 p(x,t) = c^2 \partial_x^2 p(x,t)$$

where p is the pressure, c the velocity (without source term). The space derivatives describe the interaction of different particles (volume areas). A finite difference approximation to these equations

$$p(t+dt) = 2\,p(t) - p(t-dt) + \frac{p(x+dx) - 2\,p(x) + p(x-dx)}{dx^2} c^2 dt^2$$

expresses the fact that what happens in the near future (*t*+*dt*) at some point x only depends on the presence *t,* the immediate past (*t*-*dt*), and the state of the system in the ***nearest neighbourhood*** of x, x±dx.

The art of parallel programming is identifying the part of the problem which can be efficiently parallelised.

Concepts of Parallel Comnputing

Von Neumann (serial) style: Only one processor which is executing a series of instructions, there is a logical sequential flow through the program. At any *one* time there is only *one* operation being carried out by the processor.

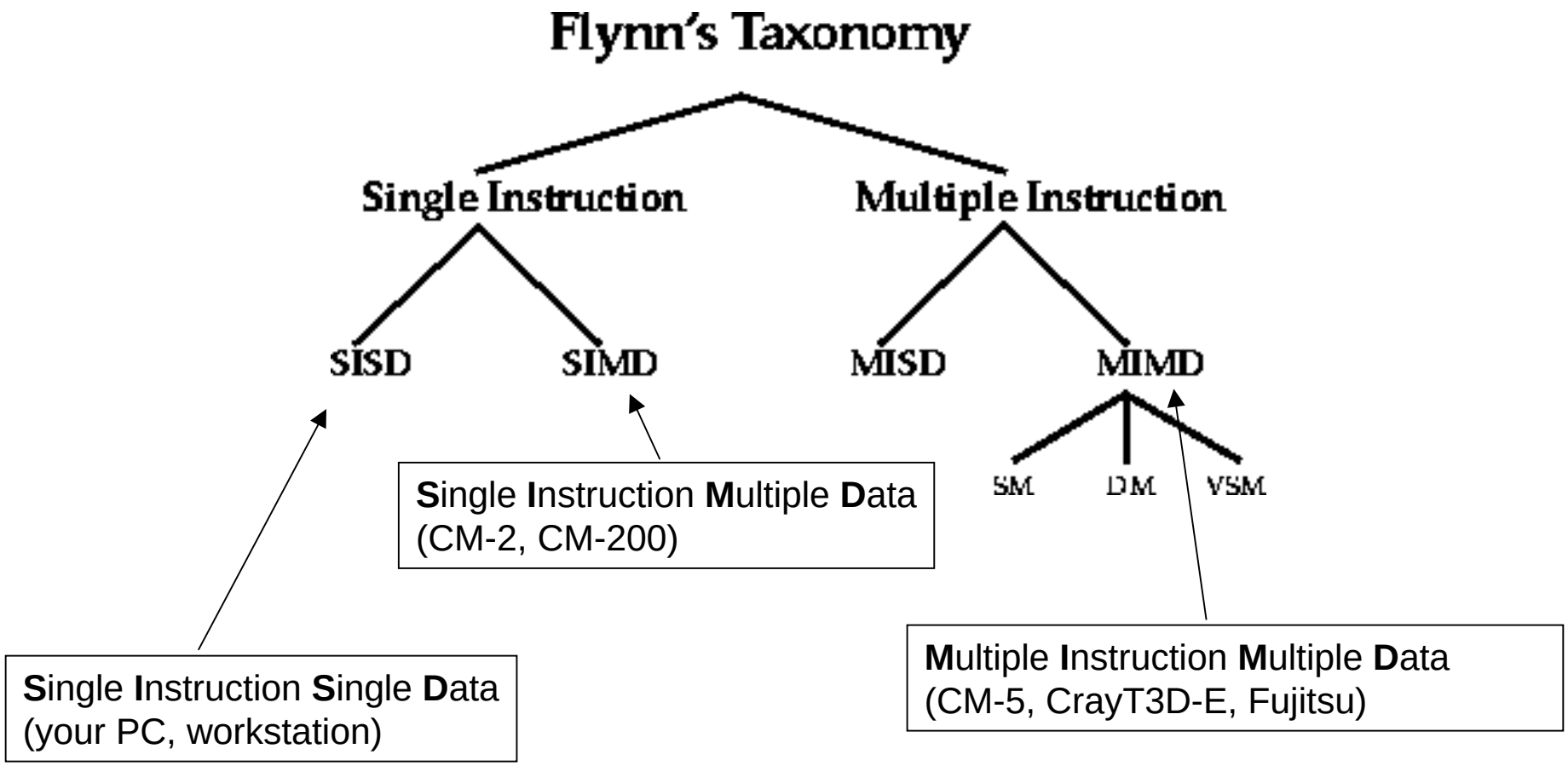
Parallel computing is aimed at producing the same result using multiple processors. The problem is divided up between a number of processors. Some aspects are

- Functional decomposition (multi-tasking)
- Data decomposition (data-parallel)
- keep all the processors busy (load-balancing)
- minimise inter-processor communication

Often there is a trade-off between communications and processing!

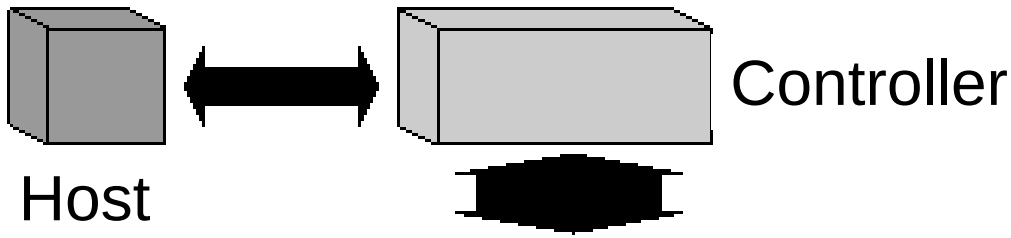
Parallel Architectures - SIMD - MIMD

Flynn's Taxonomy is a standard way of describing different architecture types

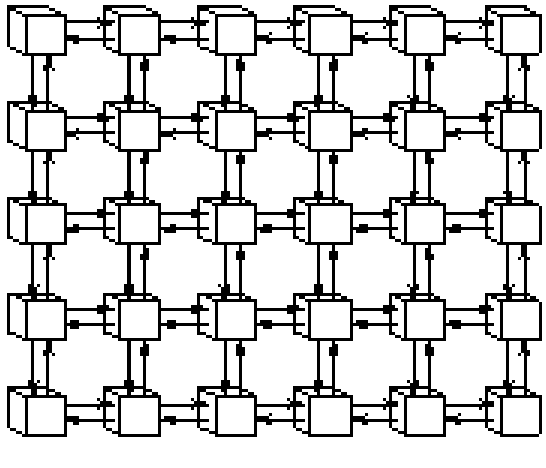


SIMD Architecture

Single **I**nstruction **M**ultiple **D**ata



e.g. massively parallel machines (CM-2), many simple processors. Each processor performs same instruction but on different data. Communication through shift operations.

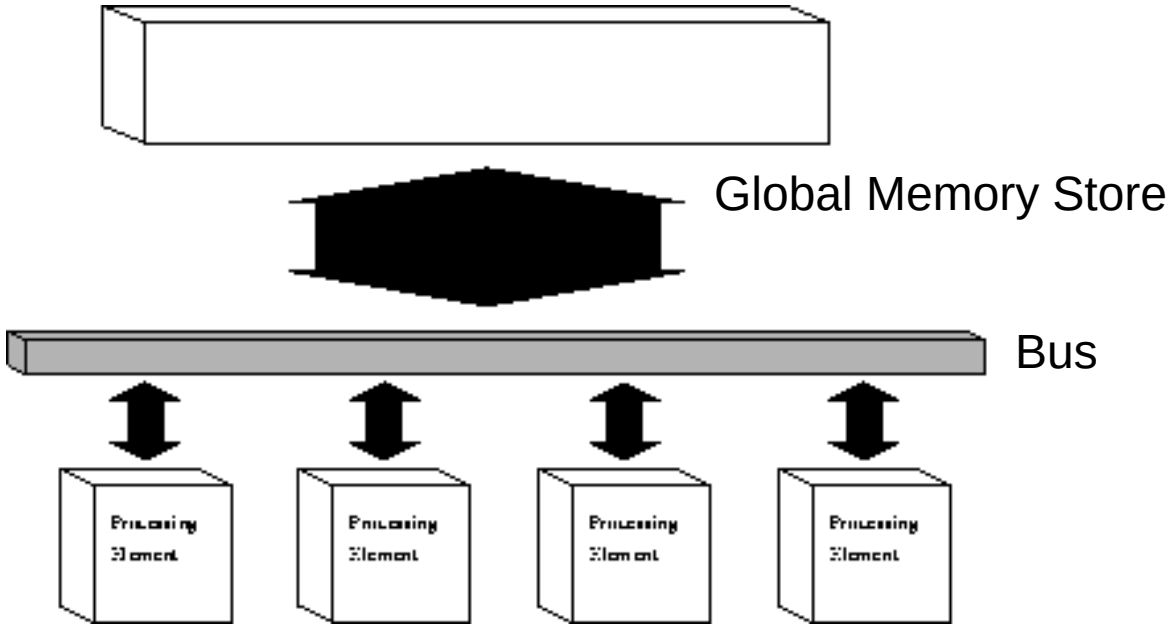


Array of processing elements

Ideal for image processing, FD calculations, cellular automata

MIMD Architecture - Shared

Multiple **I**nstruction **M**ultiple **D**ata

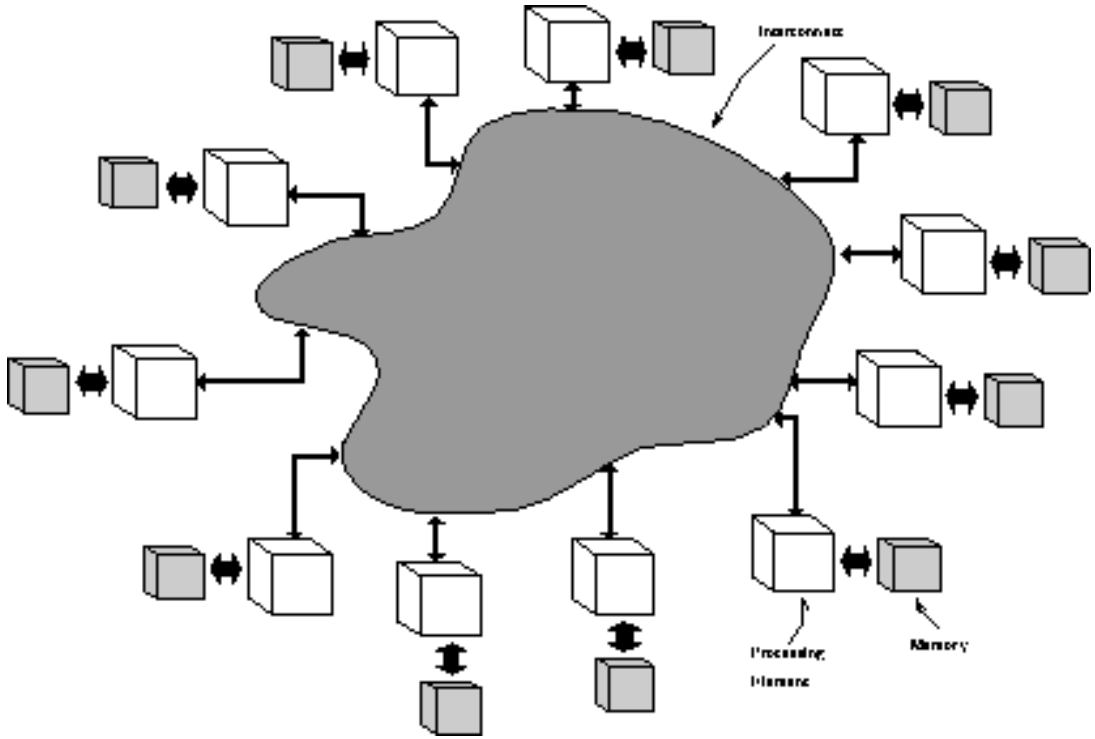


Processing elements

Shared Memory Model. Usually small number of processors with access to global memory via a **bus.** The time to access any data is the same. Problems, when a number of processors attempts to access the same memory. (e.g. SGI PowerChallenge, DEC8400)

MIMD Architecture - Distributed

Multiple **I**nstruction **M**ultiple **D**ata



Distributed Memory Model. Each processor has its own memory. Communications within a processor much faster than interprocessor communication. Drawbacks because of explicit communications between processors. More scalable than shared memory architecture. (e.g. Linux clusters).

Parallel Architecture - Outlook

Modern architectures really are mixtures of these basic types. Main problems from a user point of view:

How difficult is it to convert a serial code to a parallel machine?

How difficult is it to transport code from one parallel machine to another one?

Can I develop parallel code locally (e.g. Linux, workstation Clusters)?

How much time do I have to spend to parallelise my code, is it worth it?

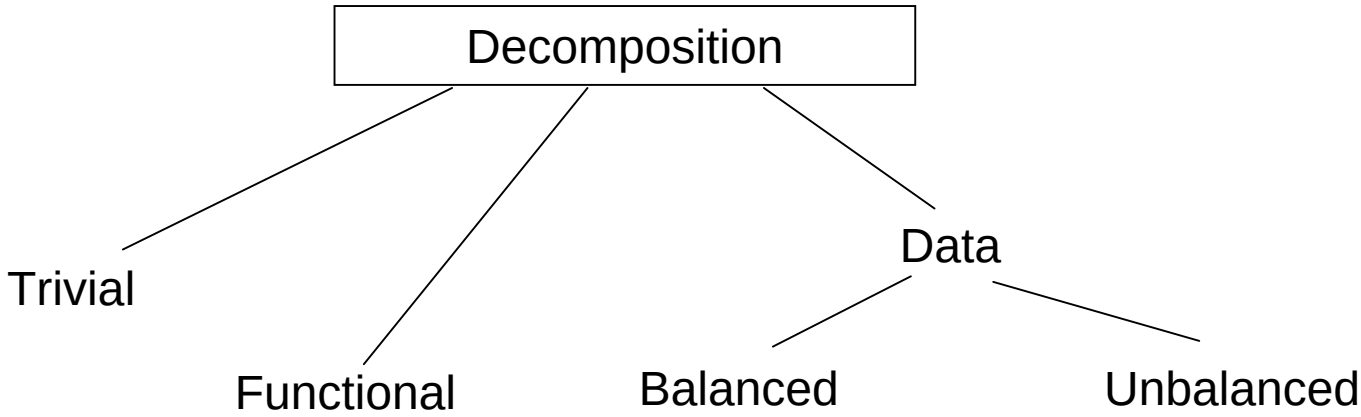
What programming model should I use (data parallel, message massing)?

Are there international standards, how stable are they?

We first have to find the potentially parallel in our problem!

Is my problem parallel ?

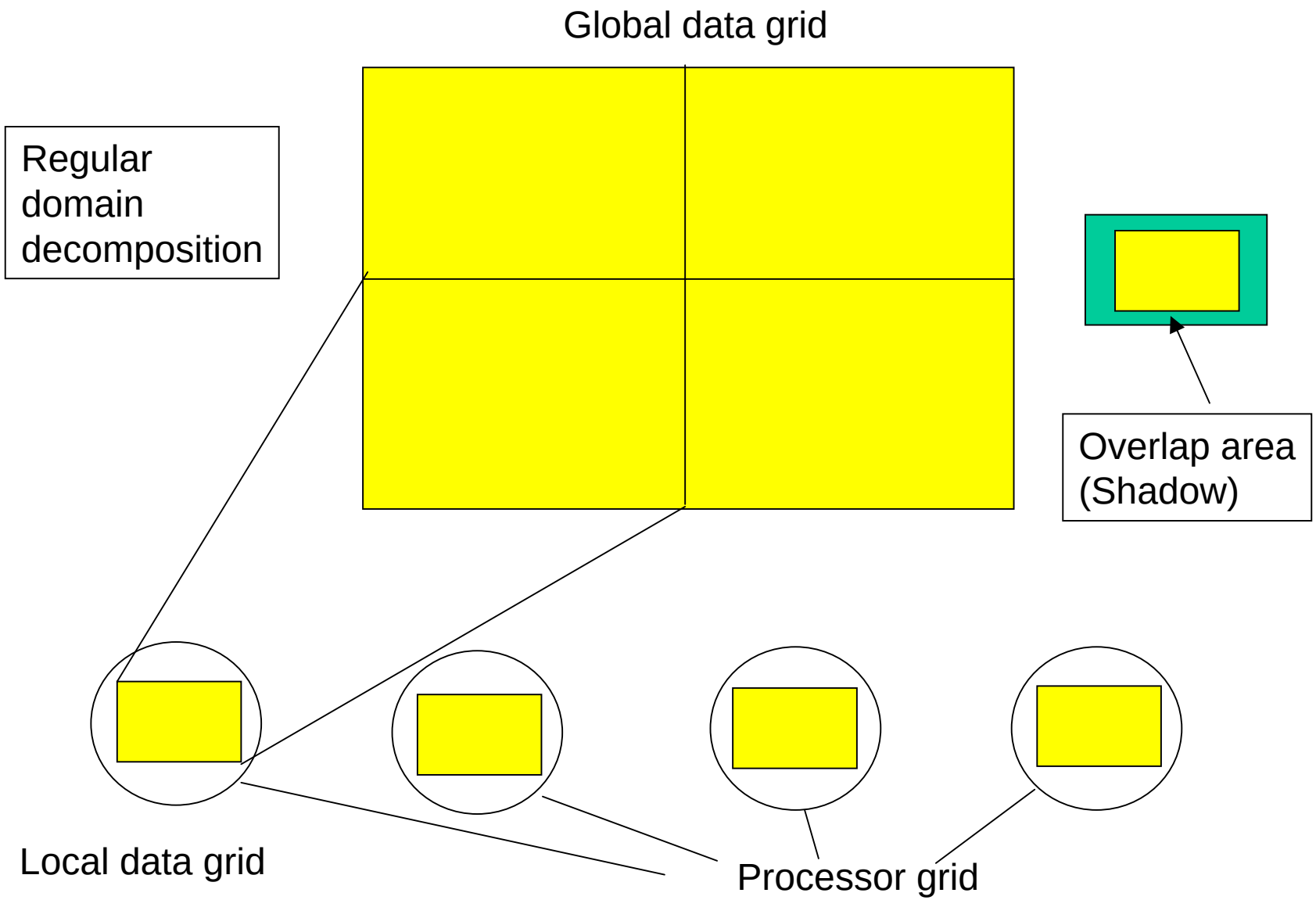
Let us assume we have a sequential code and we want to run it faster by executing in parallel. We need to decompose the problem:



Trivial: The same program needs to be run for many different input parameters. No communication is needed between processors.

Functional: A program is broken up into several subprograms with different tasks. Problems, if different subprograms take different time and synchronisation is necessary.

Data decomposition



Parallel programming

What programming concepts exist for parallel programs?

Data-parallel programming: Global memory is provided that can be directly read from and written to by every process involved in a computation.

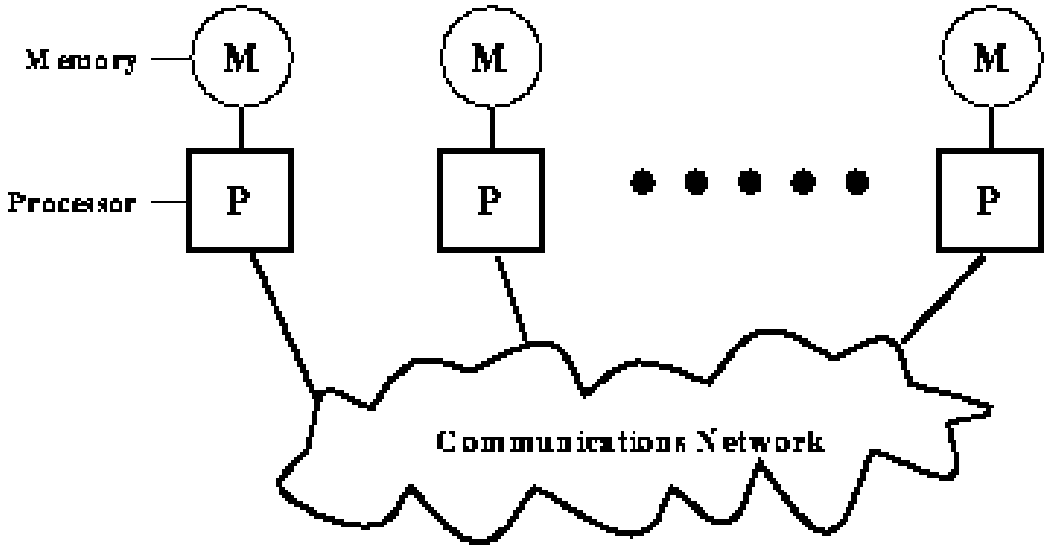
Message-passing programming: Each process has a local memory and no other process can directly read from or write to that local memory.

Distributed-memory programming: No globally addressable memory as in the data parallel model.

Usually parallel computers support all these programming models with various compilers. It is up to the programmer to decide which one suits the problem best.

Message-passing concepts

Massage passing (MP) programs are written as sequential code. However, at any point the programmer specifies the actions of a process performing part if the calculation rather than specifying the action of the entire program.



Distribute processes - synchronise - exchange data

Message-passing programs

Process A

whoami: 0

x: 0

Process B

whoami: 1

x: 0

if (whoami==0) x=x+111

Process A

whoami: 0

x: 111

Process B

whoami: 1

x: 0

Messages

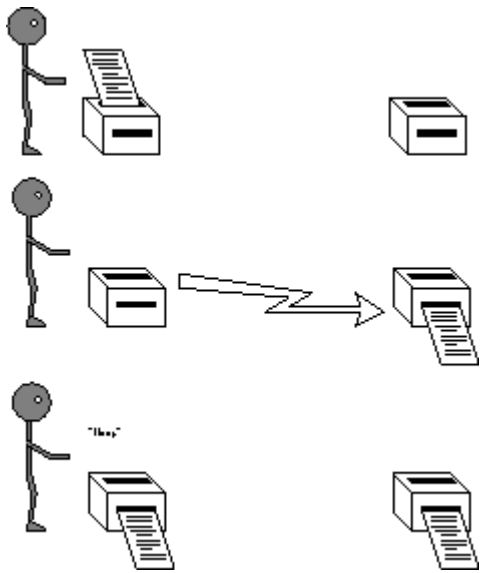
Central to the message-passing concept are messages, the following questions arise

- Which processor is sending a message?
- Where is the data on the sending processor?
- What kind of data is being sent? How much?
- Which processor(s) are receiving the message?
- Where should the data be left on the receiving processor?
- How much data is the receiving processor prepared to accept?

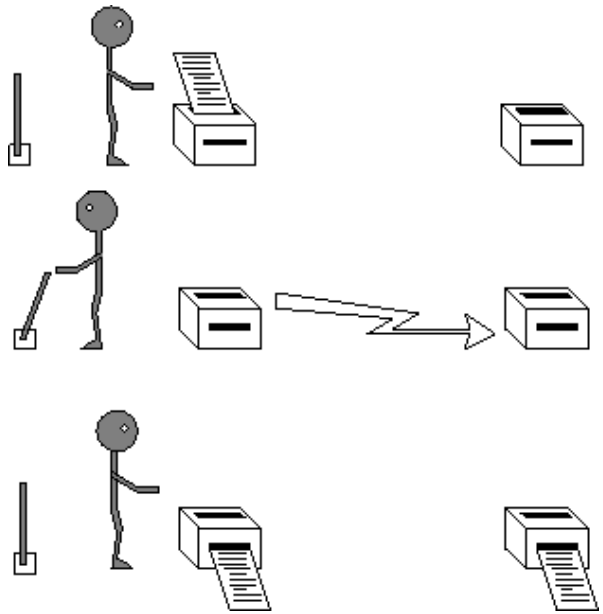
Messages - point-to-point

Send and ***receive*** required. ***Synchronous*** if communication does not complete until the message has been sent, ***asynchronous*** otherwise.

Blocking



Non-blocking

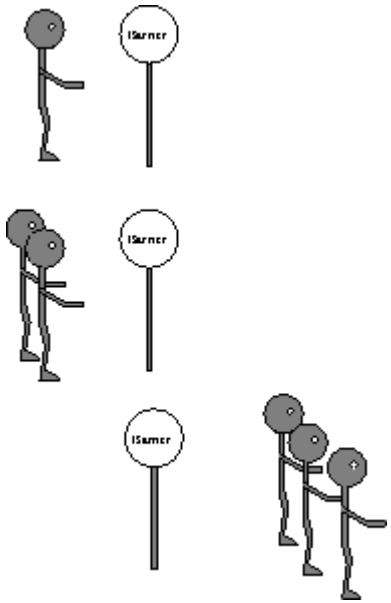


useful work can be done while message is sent

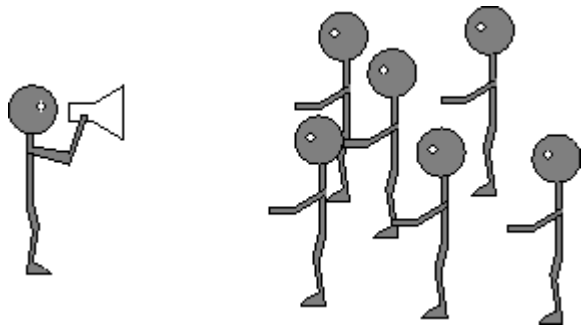
Messages - Synchronisation

A ***barrier*** operation synchronises processes (processors). No data is exchanged until all of the participating processors have called the ***barrier*** routine.

Barrier



Broadcast
one-to-many

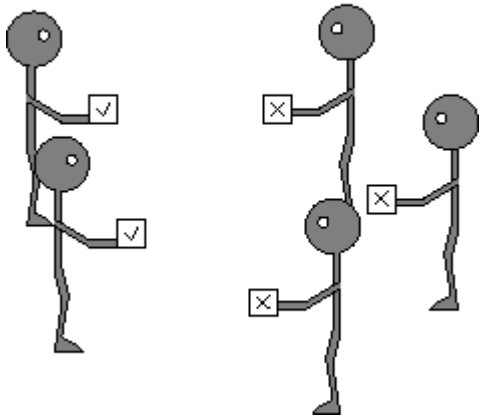


One processor sends the same message to a number of recipients with a single operation.

Messages - Reduction operation

Reduction operation reduces data from a number of processors to a single item e.g. sum all elements of the variables called x in each processor

Strike



Message Passing - Example 1

The processor with identification ***rank*** calls a subroutine with a particular input ***ifreq*** . This immitates a problem where a particular task is performed as a function of frequency in parallel.

```
      call mpi_init(ierror)

      call mpi_comm_rank(mpi_comm_world,rank,ierror)
      call mpi_comm_size(mpi_comm_world,size,ierror)


      ifreq=0
      if(rank.eq.0) call mysub(ifreq,niter)
      ifreq=1
      if(rank.eq.1) call mysub(ifreq,niter)
      ifreq=2
      if(rank.eq.2) call mysub(ifreq,niter)
      ifreq=3
      if(rank.eq.3) call mysub(ifreq,niter)
      ifreq=4
      if(rank.eq.4) call mysub(ifreq,niter)
      ifreq=5
      if(rank.eq.5) call mysub(ifreq,niter)
```

Message Passing - Example 2

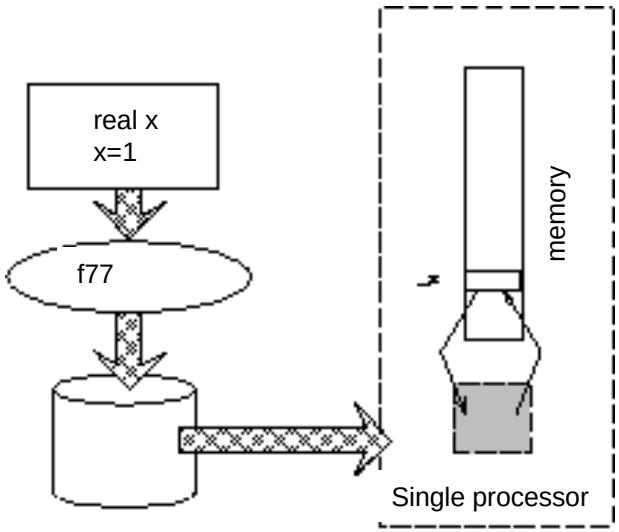
... and in a more condensed notation ...

```
      call mpi_init(ierror)

      call mpi_comm_rank(mpi_comm_world,rank,ierror)
      call mpi_comm_size(mpi_comm_world,size,ierror)


      do nproc=0,size

      if(rank.eq.nproc) then
      call mysub(nproc,niter)
      endif

      enddo
```

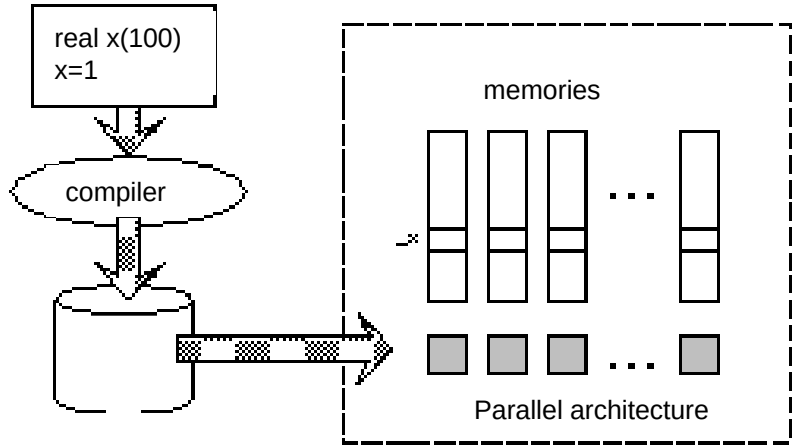
... each processor is performing the mysub routine on different data.

Data parallel concepts

Sequential (scalar)



Parallel (array)



Arrays are the prototype of parallel data! Parallel data are accessed in the source program as though it was shared by all processors. Data parallel programs may not even need directives (e.g. CM Fortran).

Data parallel languages

Unfortunately the supercomputer industry was not able to come up with a stable portable standard for data parallel applications. Some attempts were: Connection Machine Fortran (CMF), High Performance Fortran (HPF), Cray Fortran (CRAFT). Fortran90 was developed with the prospects of data parallel hardware.

Array operations:

F77

```
real x(64), b(64)
do i=1,64
    a(i)=2.0
    b(i)=b(i)*a(i)
enddo
```

F90

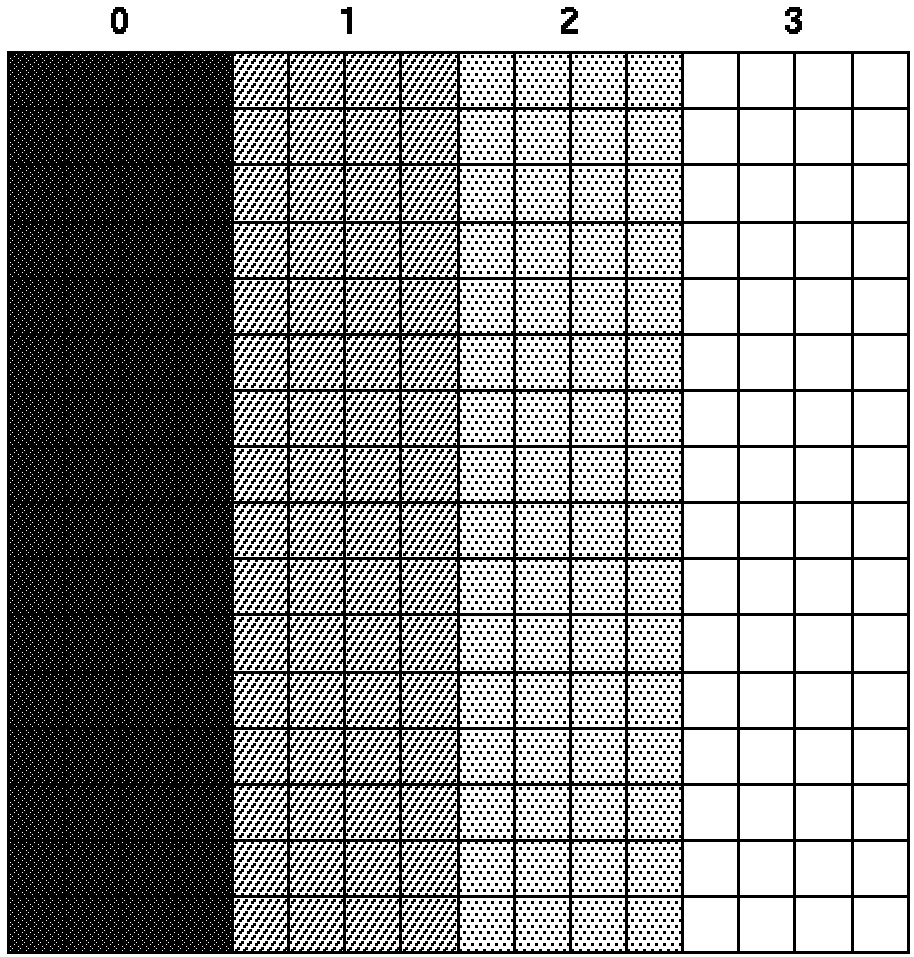
```
real x(64), b(64)

a=2.0
b=b*a
```

The data parallel compiler will distribute the work so that memory references are primarily local (which is trivial in this case, if arrays a and b have the same ***layout***, i.e. element ij of array a is in the same processor as element ij of array b).

Data parallel - data distribution

Layout (distribution)
of parallel array

a(16,16)
a(*,block) -> HPF
a(:serial,:news) -> CMF

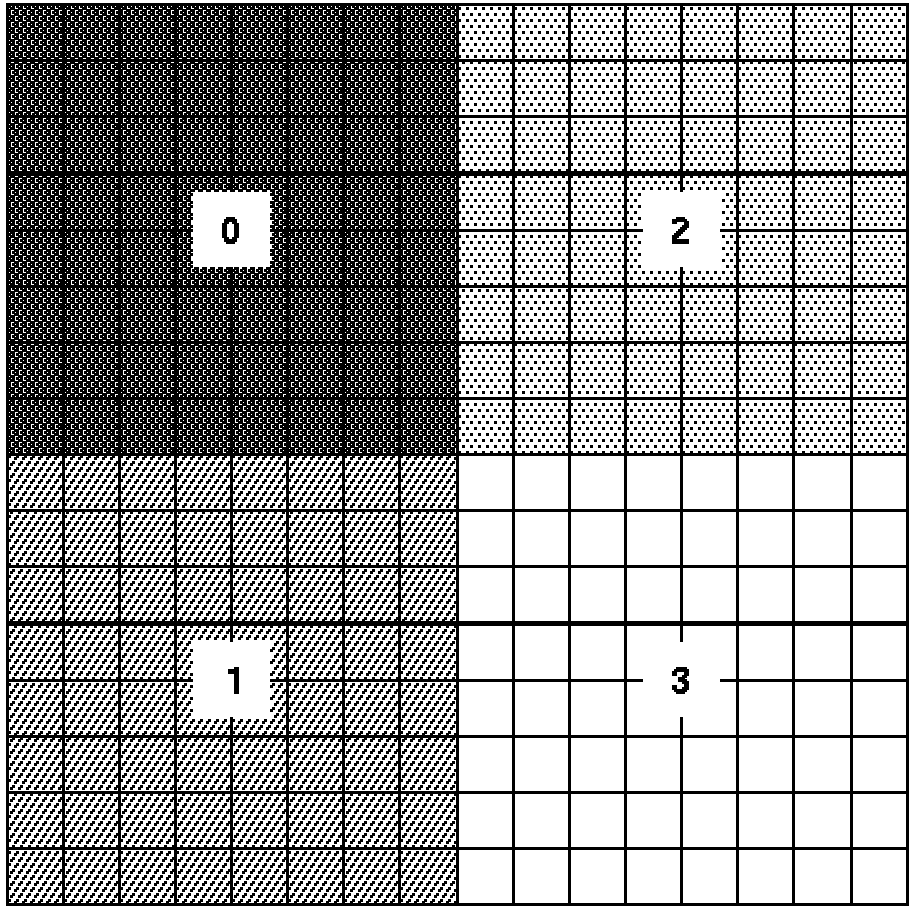


One dimension (e.g. a(:,1)) is known in one processor.

Data parallel - data distribution

Layout (distribution)
of parallel array

a(16,16)
a(block,block) -> HPF
a(:news,:news) -> CMF

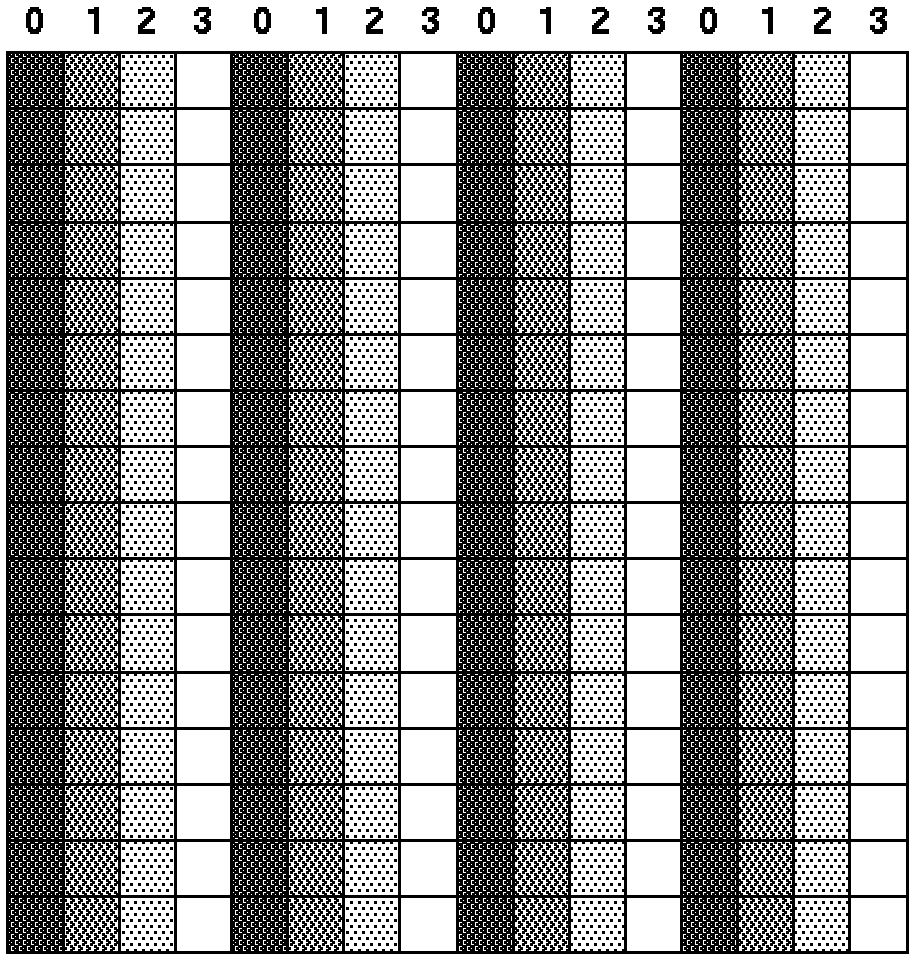


Whole array is divided up in *number_of_processors* parts

Data parallel - cyclic distribution

Layout (distribution)
of parallel array

a(16,16)
a(*,cyclic) -> HPF



This layout is to be preferred when global communications are necessary.

Data parallel - Example 1

Relaxation problem as data parallel algorithm

F77

```
DO k = 1, number_of_iterations
 DO i = 2, n-1              ! Update non-edge
  DO j = 2, n-1             ! elements only
   temp(i, j) = (slab(i, j-1)+slab(i-1, j)+slab(i+1, j)+slab(i, j+1))/4
  END DO
 END DO
 DO i = 2, n-1
  DO j = 2, n-1
   slab(i, j) = temp(i, j)
  END DO
 END DO
END DO
```

F90

```
DO k = 1, number_of_iterations
  FORALL (i=2:n-1, j=2:n-1)      ! Non-edge elements only
    slab(i, j) = (slab(i, j-1)+slab(i-1, j)+slab(i+1, j)+slab(i, j+1))/4
  END FORALL
END DO
```

Note the necessity for an auxiliary array with F77

Data parallel - Example 2

```
          subroutine pder2d(h,f,dx,nx,nz,dir,dim)

          real, dimension(nx,nz),intent(out) :: h
          real, dimension(nx,nz),intent(in) :: f
!hpf$   distribute *(*,block), shadow(0,4) :: f
!hpf$   align h with f

          integer dir,dim
          real dx


          if (dim == 1) then
          if ( dir == -1 )then

                    forall (i=2:nx-1,j=2:nz-1)
                    h(i,j)=(f(i,j)-f(i-1,j))/dx
                    end forall

          else

                    forall (i=2:nx-1,j=2:nz-1)
                    h(i,j)=(f(i+1,j)-f(i,j))/dx
                    end forall

          endif

          else

     … and so on …

      end subroutine pder2d
```

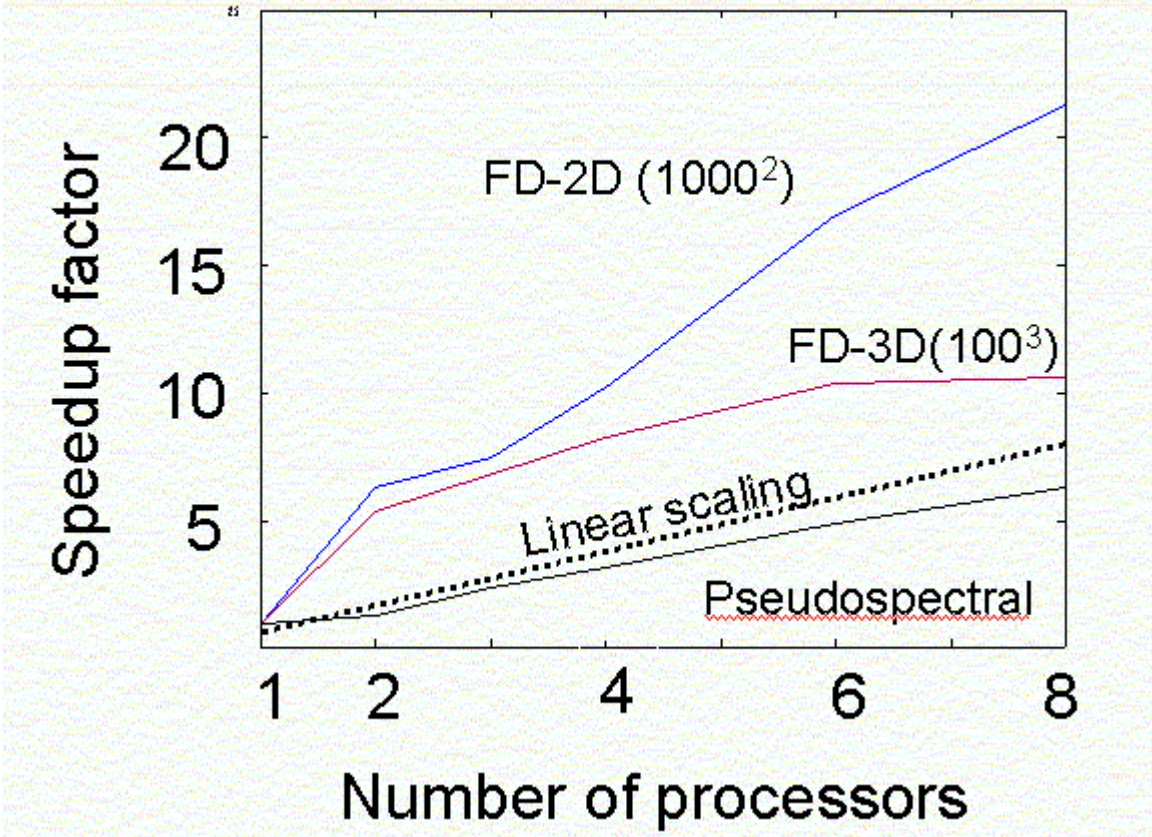
Layout (distribution) of arrays

Global addressing but parallel execution of forall statement

Finite difference first derivative with High-Performance Fortran (HPF) language and directives.

Data parallel - Scaling

Scaling of FD and PS derivatives on DEC8400-625
with 8 processors (shared memory, symmetric processors)



superlinear scaling through efficient cache-memory handling

Supercomputing - Summary

Parallel computing plays an increasingly important role in all realistic numerical modeling situations (weather prediction, fluid dynamics, seismology, cosmology, banking, etc.)

The main programming models are **message passing** (e.g. MPI, PVM) and **data parallel** languages (e.g. HPF, CRAFT).

At present, the most stable and portable language seems to be the machine independent **MPI** standard. MPI offers the option to run programs in parallel on any area network (e.g. workstation-, Linux-clusters).

Many time-dependent problems in **geophysics** (e.g. mantle convection, Earth's magnetic field, seismic wave propagation, etc.) are highly data-parallel problems when solved using local differential operators (e.g. **finite differences**).