Programming with MPI

Parallel programming

Jan Thorbecke

Acknowledgments

- This course is partly based on the MPI courses developed by
  - Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
    http://www.hlrs.de/home/
    https://www.epcc.ed.ac.uk

H L R S
High-Performance Computing Center | Stuttgart

- CSC – IT Center for Science Ltd.
  - https://www.csc.fi
  - https://research.csc.fi

CSC – IT CENTER FOR SCIENCE
C S C

- http://mpitutorial.com

Contents

Amdahl's Law

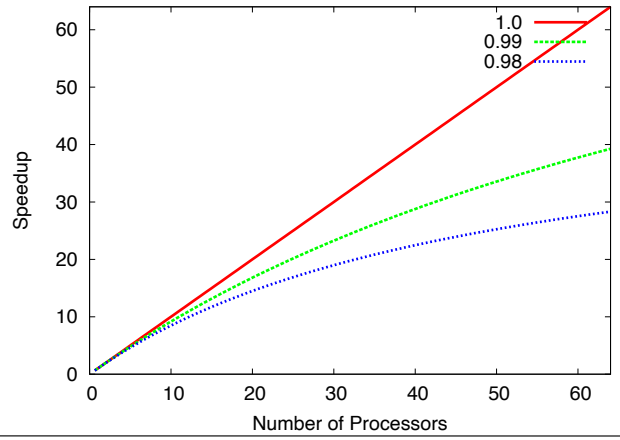
- Describes the relation between the parallel portion of your code and the expected speedup

$$speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

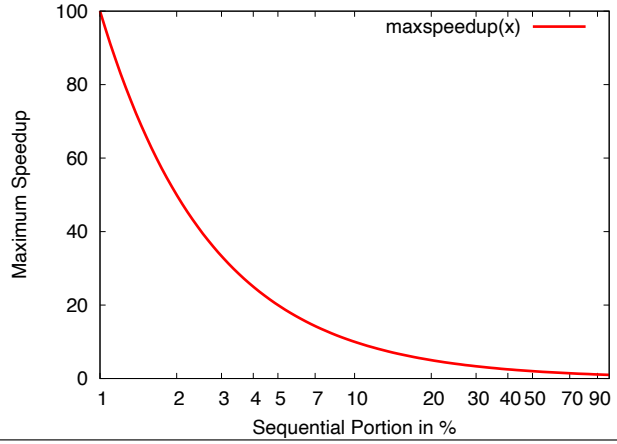
- P = parallel portion
- N = number of processors used in parallel part

- P/N is the ideal parallel speed-up, it will always be less

Amdahl's Law

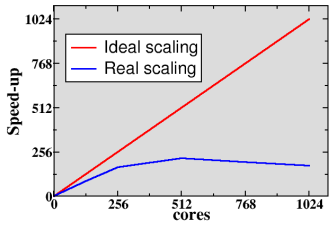


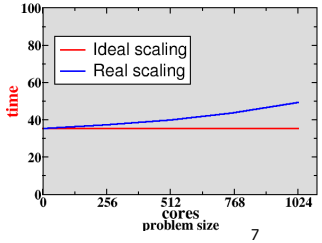
Amdahl's Law



Concepts

- Strong parallel scaling
  - constant problem size
  - execution time decreases in proportion to the increase in the number of cores
  - shortens the time to solve a problem
- Weak parallel scaling
  - increasing problem size
  - execution time remains constant when number of cores increases in proportion to the problem size
  - enables to solve larger problems

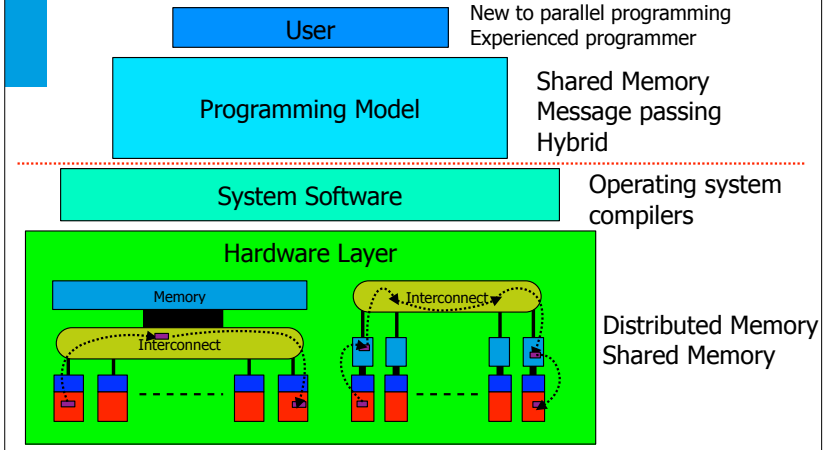




Parallel Programming Models

- Shared Memory
  - tasks share a common address space, which they read and write asynchronously.
- Threads
  - a single process can have multiple, concurrent execution paths. Example implementations: POSIX threads & **OpenMP**
- Message Passing
  - tasks exchange data through communications by sending and receiving messages. Example: MPI
- Data Parallel languages
  - tasks perform the same operation on their partition of work. Example: Co-array Fortran (CAF), Unified Parallel C (UPC), Chapel
- Hybrid
  - MPI + OpenMP

Programming Models



Parallel Programming Concepts

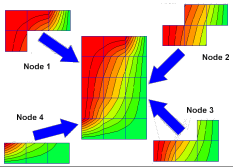
- Work distribution
  - Which parallel task is doing what?
  - Which data is used by which task?
- Synchronization
  - Do different parallel tasks meet?
- Communication
  - Is communication between parallel parts needed?
- Load Balance
  - Does every task has the same amount of work?
  - Are all processors of the same speed?

Distributing of Work and/or Data

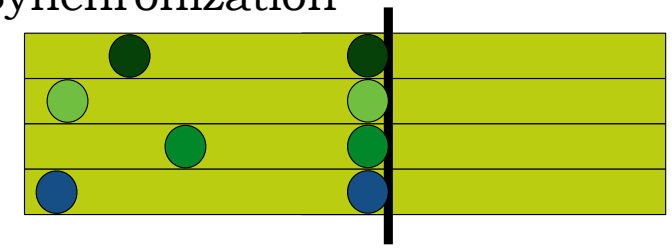
- Work decomposition
  - based on loop counter
- Data decomposition
  - all work for a local portion of the data is done by the local processor
- Domain decomposition
  - decomposition of work and data

```
do i=1,100
  1: i=1,25
  2: i=26,50
  3: i=51,75
  4: i=76,100

A(1:10,1:25)
A(1:10,26:50)
A(11:20,1:25)
A(11:20,25:50)
```



Synchronization



- Synchronization: all parallel processes must reach this point
  - causes overhead
  - idle time, when not all tasks are finished at the same time: load imbalance

Communication

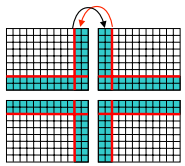
- communication is necessary on the boundaries

```
do i=2,99
   b(i) = b(i) + h*(a(i-1)-2*a(i)+a(i+1))
end do

e.g. b(26) = b(26) + h*(a(25)-2*a(26)+a(27))
```

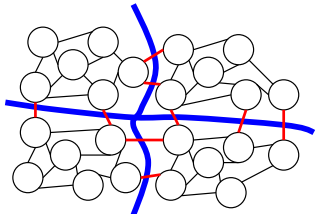
A(1:25)
A(26:50)
A(51:75)
A(76:100)

- domain decomposition



Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to:
  - less parallelism than processors
  - unequal sized tasks together with too little parallelism
  - unequal processors

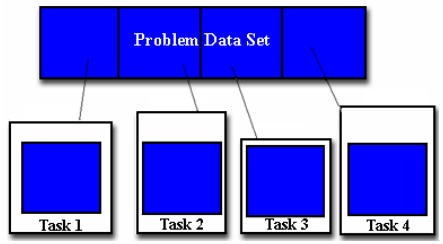


Examples of work distribution

- Domain Decomposition
- Master Worker
- Task Decomposition

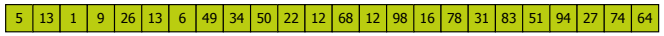
Domain Decomposition

- First, decide how data elements should be divided among processors
- Second, decide which tasks each processor should be doing



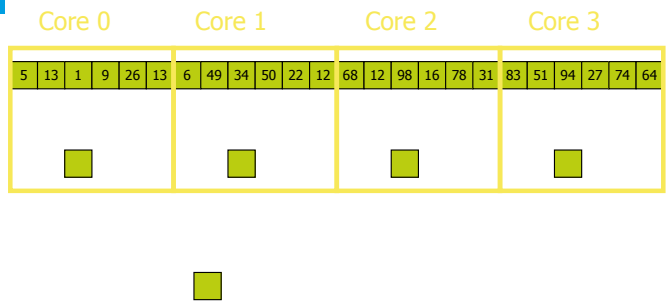
Domain Decomposition

Find the largest element of an array



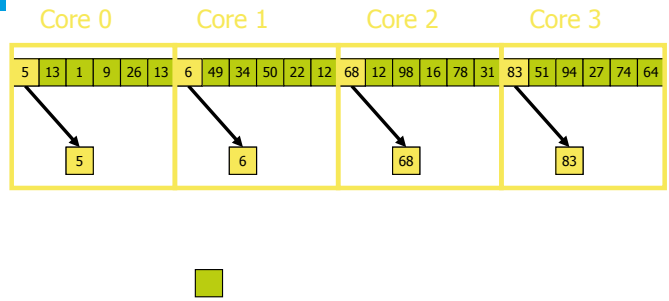
Domain Decomposition

Find the largest element of an array



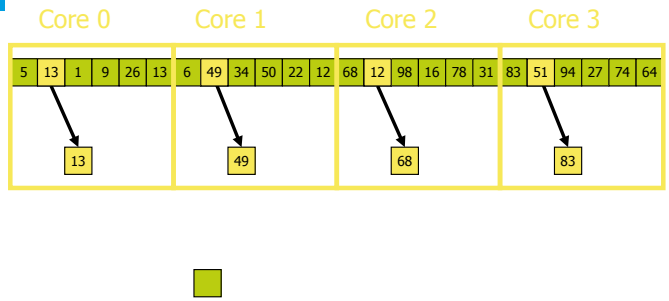
Domain Decomposition

Find the largest element of an array



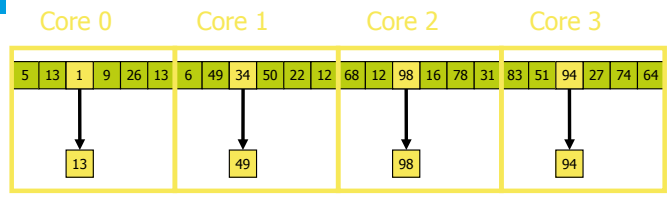
Domain Decomposition

Find the largest element of an array



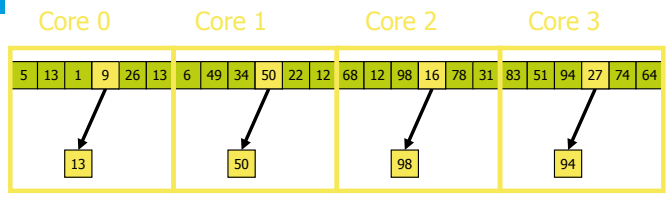
Domain Decomposition

Find the largest element of an array



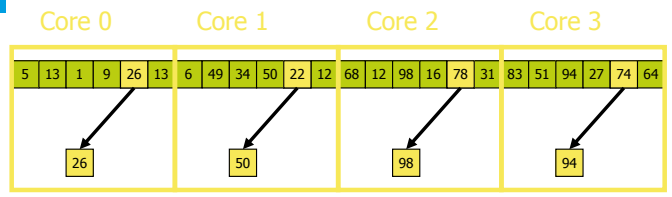
Domain Decomposition

Find the largest element of an array



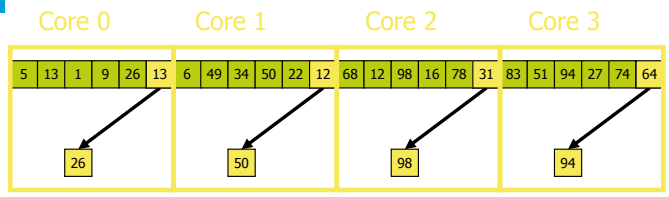
Domain Decomposition

Find the largest element of an array



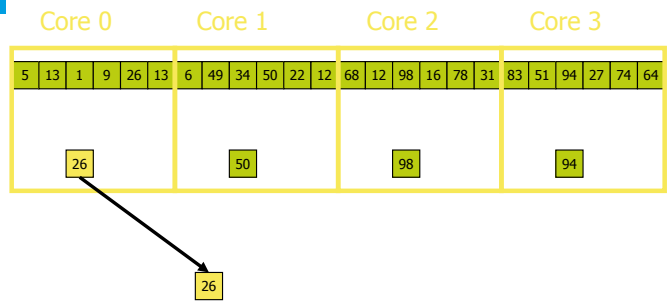
Domain Decomposition

Find the largest element of an array



Domain Decomposition

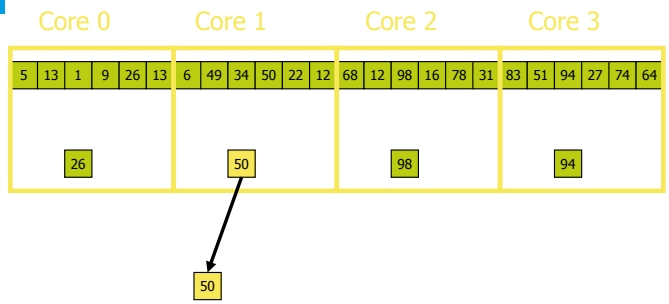
Find the largest element of an array



TU Delft

Domain Decomposition

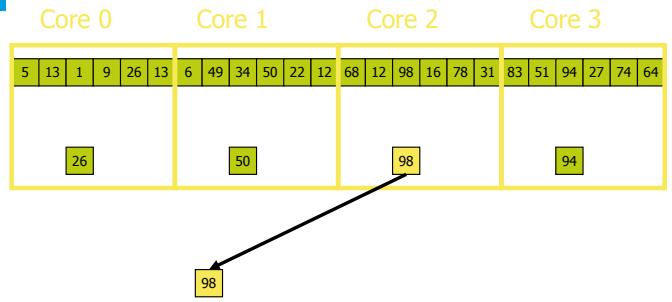
Find the largest element of an array



TU Delft

Domain Decomposition

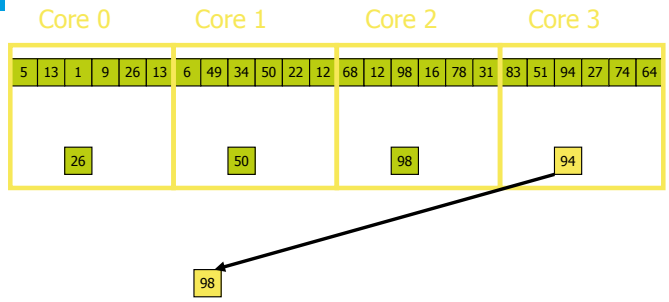
Find the largest element of an array



TU Delft

Domain Decomposition

Find the largest element of an array

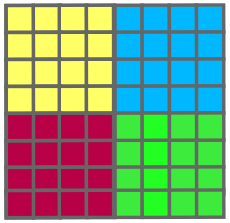


TU Delft

Domain decomposition of sum

- Data is distributed to processor cores
- Each core performs (nearly) identical tasks with different data
- Example: summing the elements of an 2 D array

core 1: ∑= core 2: ∑=

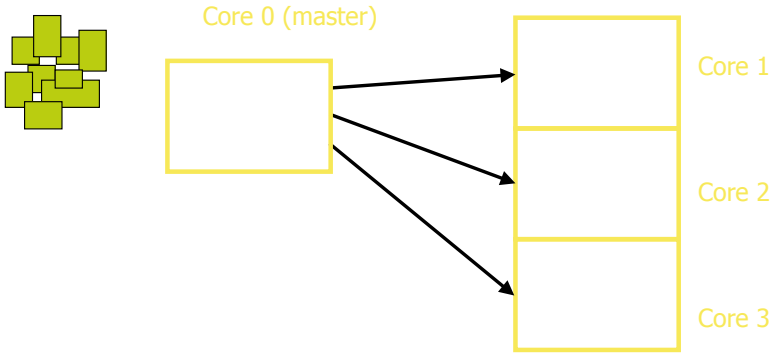


core 3: ∑= core 4: ∑=

- Each core sums it's part of the array
- The individual sums have to be combined in the end

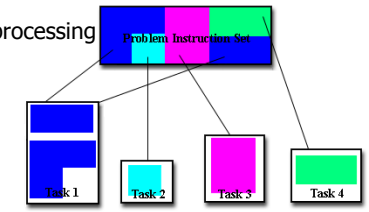
Master Worker

Get a pile of work done



Task/Functional Decomposition

- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Divide computation based on natural set of independent tasks
  - Assign data for each task as needed
- Example: pipeline seismic data pre-processing
  - static-correction
  - deconvolution
  - nmo correction
  - stacking
  - ....



OpenMP™

What Is OpenMP?

- Compiler directives for multithreaded programming
- Easy to create threaded Fortran and C/C++ codes
- Supports data parallelism model
- Portable and Standard
- Incremental parallelism
  - Combines serial and parallel code in single source

Directive based

- Directives are special comments in the language

– Fortran fixed form: `!$OMP, C$OMP, *$OMP`

– Fortran free form: `!$OMP`

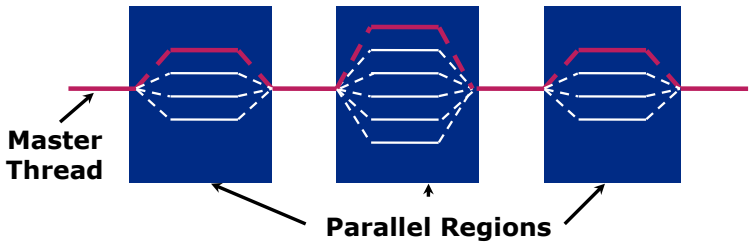
Special comments are interpreted by OpenMP compilers

```
      w = 1.0/n
      sum = 0.0
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:sum)
      do I=1,n
        x = w*(I-0.5)
        sum = sum + f(x)
      end do
      pi = w*sum
      print *,pi
      end
```

Comment in Fortran but interpreted by OpenMP compilers

Programming Model

- Fork-join parallelism:
  - Master thread spawns a team of threads as needed
  - Parallelism is added incrementally: the sequential program evolves into a parallel program

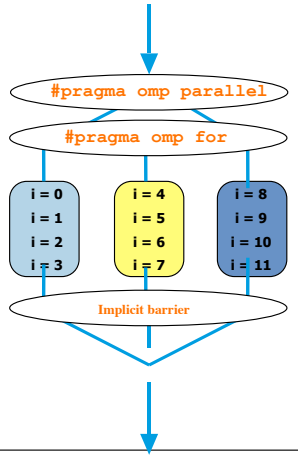


Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; i++)
      c[i] = a[i] + b[i]
```

Threads are assigned an independent set of iterations

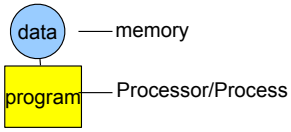
Threads must wait at the end of work-sharing construct



MPI

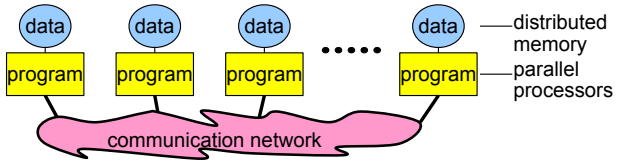
The Message-Passing Programming model

- Sequential Programming



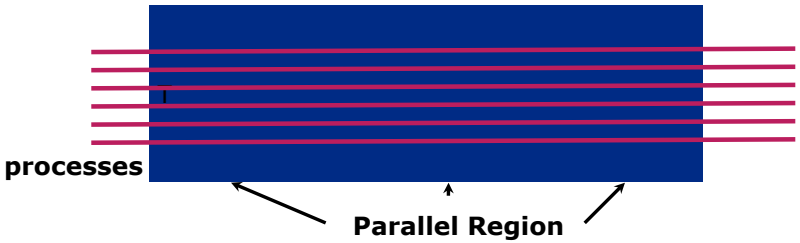
A processor may run many processes

- Message-Passing Programming



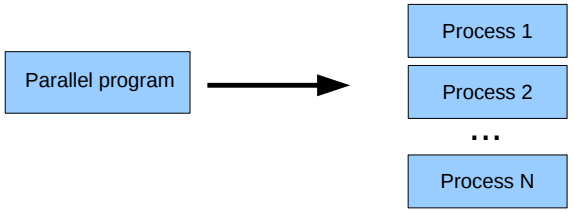
Programming Model

- Explicit parallelism:
  - All processes starts at the same time at the same point in the code
  - Full parallelism: there is no sequential part in the program



Execution model

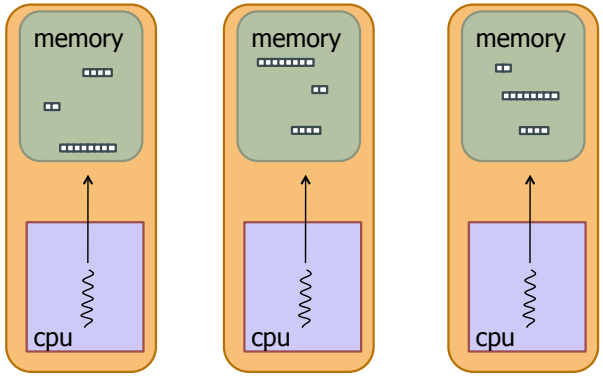
- Parallel program is launched as set of **independent, identical processes**



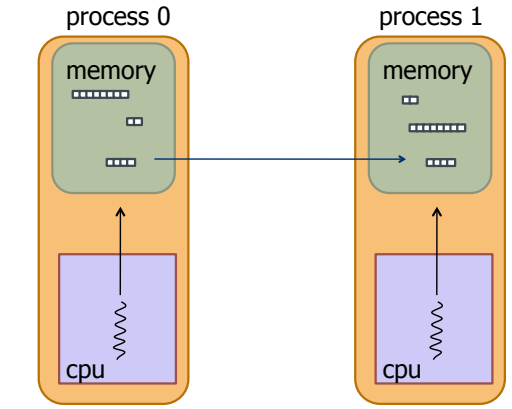
- All the processes contain the same program code and instructions
- Processes can reside in different nodes or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, aprun, poe, ...

MPI

processes



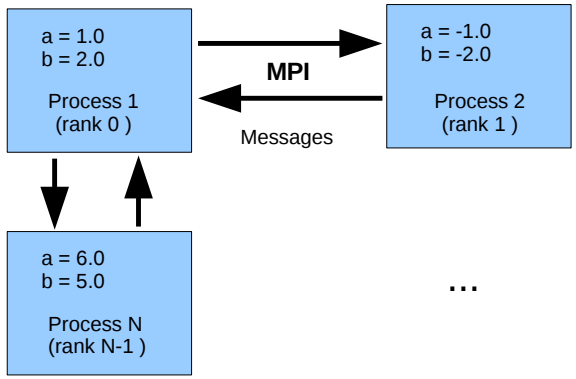
MPI




```
MPI_Send(a,...,1,…)  MPI_Recv(a,...,0,…)
```


Data Model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



MPI advantages

- Mature and well understood
  - Backed by widely-supported formal standard (1992)
  - Porting is "easy"
- Efficiently matches the hardware
  - Vendor and public implementations available
- User interface:
  - Efficient and simple
  - Buffer handling
  - Allow high-level abstractions
- Performance

MPI learning

- MPI 3.1 includes many features beyond message passing

- Execution control environment depends on implementation

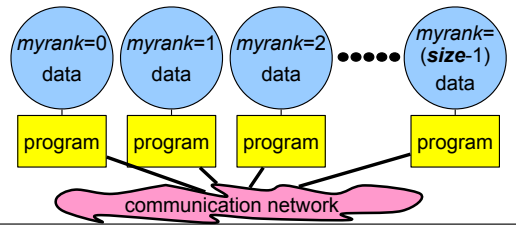
Basic Concepts

Work Distribution

- All processors run the same executable.
- Parallel work distribution must be explicitly programmed into the algorithm:
  - domain decomposition
  - master worker

Data and Work Distribution

- To communicate together mpi-processes need identifiers: rank = identifying number
- all distribution decisions are based on the rank
  - i.e., which process works on which data

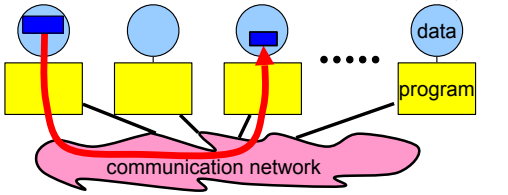


Message passing

- Point-to-Point
- Requires explicit commands in program
  - Send, Receive
- Must be synchronized among different processors
  - Sends and Receives must match
- Multi-processor communications
  - e.g. broadcast, reduce

Message passing

- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
  - sending process – receiving process } i.e., the ranks
  - source location – destination location
  - source data type – destination data type
  - source data size – destination buffer size

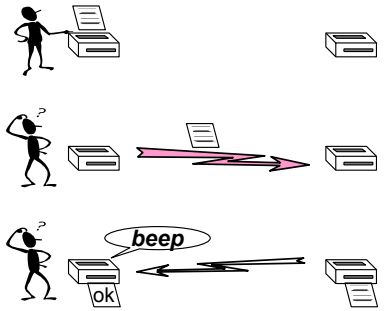


Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communications
  - **synchronous** send
  - buffered = **asynchronous** send

Synchronous Sends fax(?)

- The sender gets an information that the message is received.
- Analogue to the beep or okay-sheet of a fax.



Synchronous Sends in

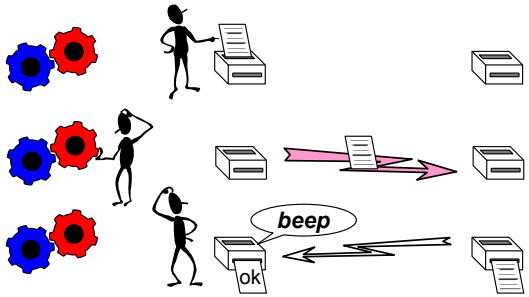
- Check marks will appear next to each message you send. Here's what each one indicates:
  - ✓ The message was successfully sent.
  - ✓✓ The message was successfully delivered to the recipient's phone or any of their linked devices.
  - ✓✓ The recipient has read your message.

Blocking Operations

- Some sends/receives may block until another process acts:
  - synchronous send operation blocks until receive is issued;
  - receive operation blocks until message is sent.
- Blocking subroutine returns only when the operation has completed.
- Don't confuse this with blocking messages in what's app !

Non-Blocking Operations

- Non-blocking operations return immediately and allow the sub-program to perform other work.

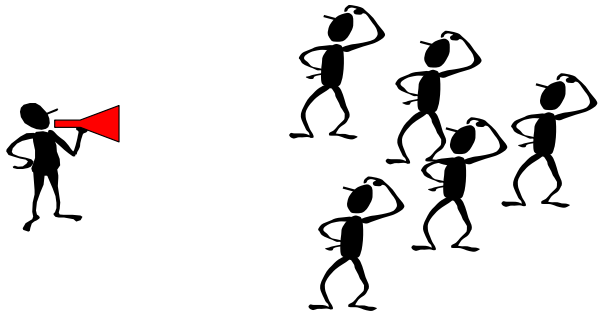


Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms

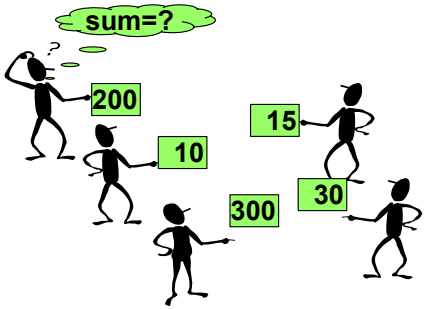
Broadcast

- A one-to-many communication.



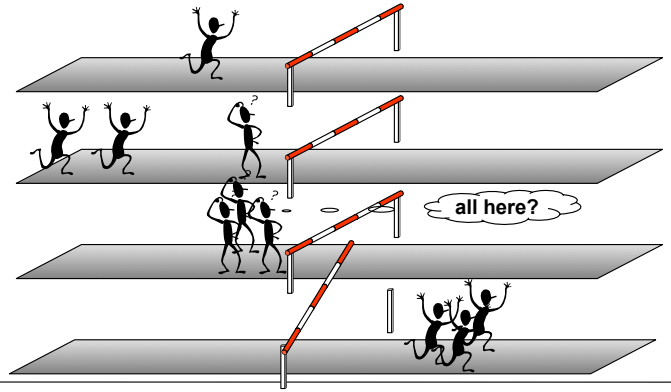
Reduction Operations

- Combine data from several processes to produce a single result.



Barriers

- Synchronize processes.



End of brief overview