Multiprocessors 2014/2015

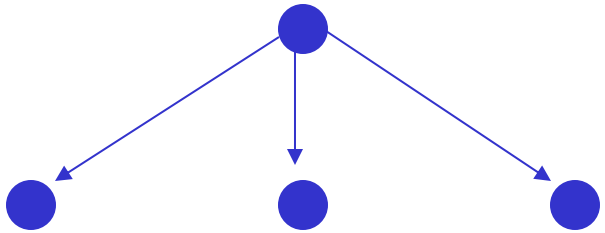
Abstractions of parallel machines
part 2: language/middleware abstraction

Johan Lukkien

Overview

- Problem statement
- Abstraction
- Operating system support
- Language / middleware support

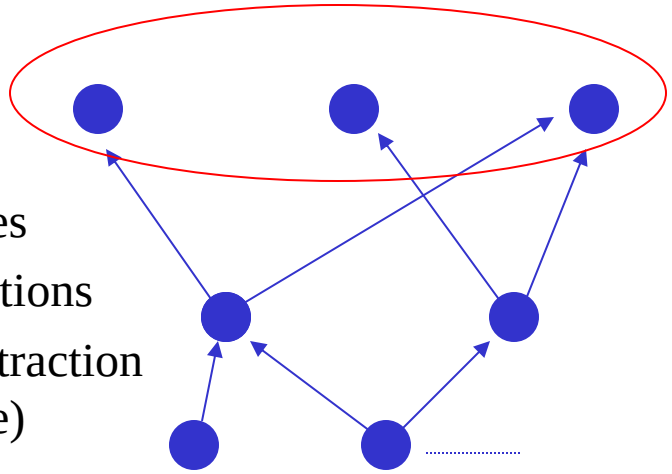
This is what we do....



problem

designs [data,functional,result]

mapping



Programming model: set of concepts, primitives and combinators to express a computation

dots: alternatives

arrows: abstractions

(no arrow: abstraction not possible)

machine model

(e.g. shared/distributed memory)

machine(s)

Concurrency at language level

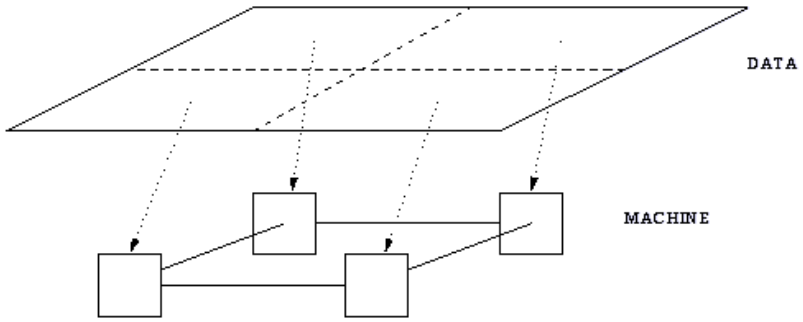
- Implicit parallelism:
  - recognize the parallelism in the program, extract and map
  - provide parallel implementation of the programming model
  - Example: parallel prolog, parallelizing compilers, some functional languages, Google MapReduce

- Explicit parallelism:
  - add concepts and primitives to the language (or library)
    - Data Parallel (e.g. CUDA)
    - Shared Memory Parallel
    - Bulk Synchronous Parallel [though this might be regarded as implicit as well]
    - Message Passing
  - Typically, this is a *framework*
    - API, compiler, run-time system, tools

Data parallel

- A large data structure is partitioned across the processors
  - (e.g., a matrix, a simulation grid)

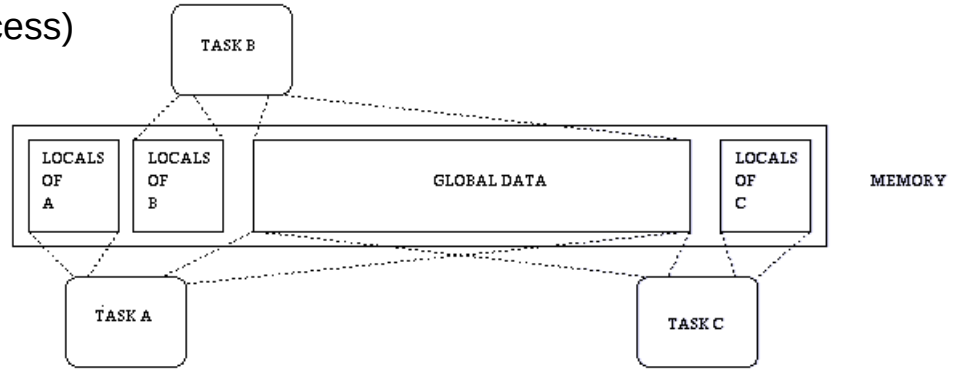
- The computation is the same (single program) for all processors
  - operating on different data
  - needing information from only a few neighboring processors



- Parallelism and communication (almost) implicit
  - typically, instructions for data distribution (i.e., the storage and computational responsibilities) and for pointing out concurrency
  - example: HPFortran, OpenMP, dedicated languages, CUDA

Shared memory parallelism

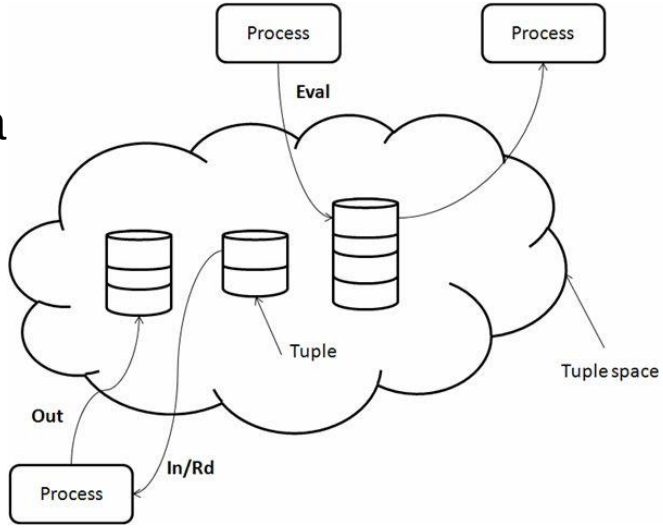
- The program consists of a series of tasks
  - executed in parallel
  - using a common memory
- Tasks may have local variables
  - active objects
- Communication and synchronization goes through the memory
  - using special primitives
    - shmem library of Cray/SG
      (symmetric hierarchical memory access)
    - POSIX 1003.b (multithreading)
    - JAVA



Shared data space (LINDA)

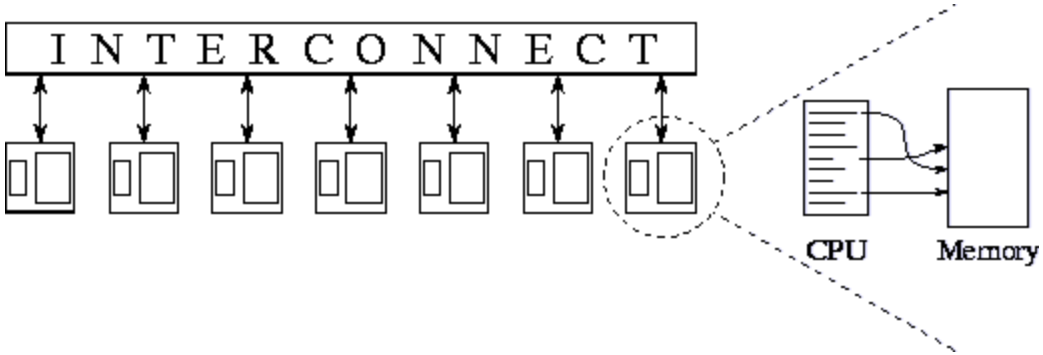
- The program consists of a set of independent workers that share a global space of tuples

- Operations
  - **in (expr):**
    - atomically reads and removes—consumes—a tuplespace that matches *expr*
  - **rd (expr):**
    - non-destructively reads from tuplespace
  - **out (tuple):**
    - produces a tuple, writing it into tuplespace
  - **eval:**
    - creates new processes to evaluate tuples, writing the result into tuplespace



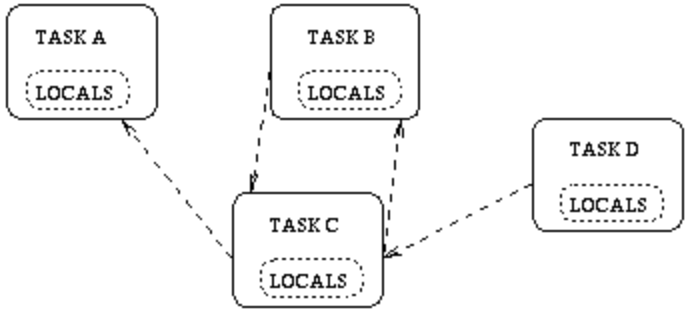
Bulk Synchronous Parallel

- The computation consists of a series of ‘supersteps’
  - computation and communication
    - communication phase: a global rearrangement of the data
    - typically via a global *barrier synchronization*
- Explicit notion of local memory
- One task per processor
- BSP-lib available (http://www.bsp-worldwide.org/, latest news in 2012)



Message passing

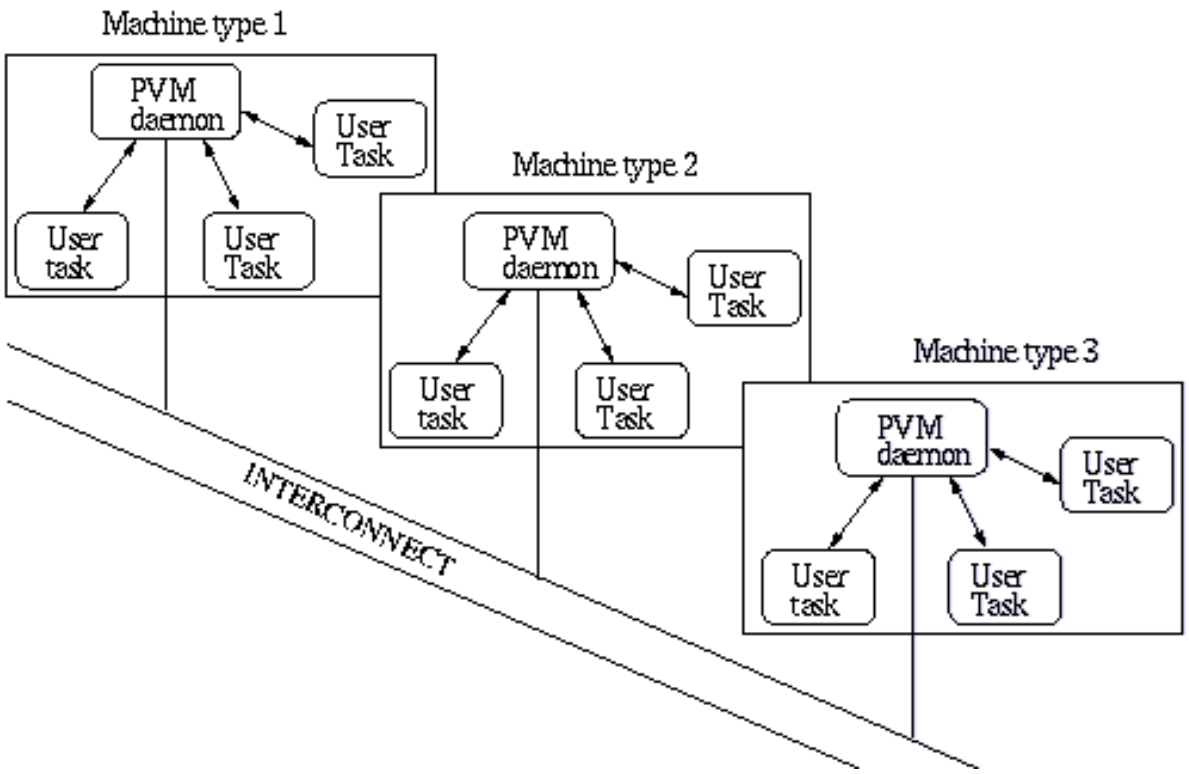
- Program: a collection of independently executed tasks
  - communicating by sending messages
    - sent to named tasks
    - or to named channels (two connected ports)
- No shared memory
- No restrictions on execution.



PVM

Parallel Virtual Machine

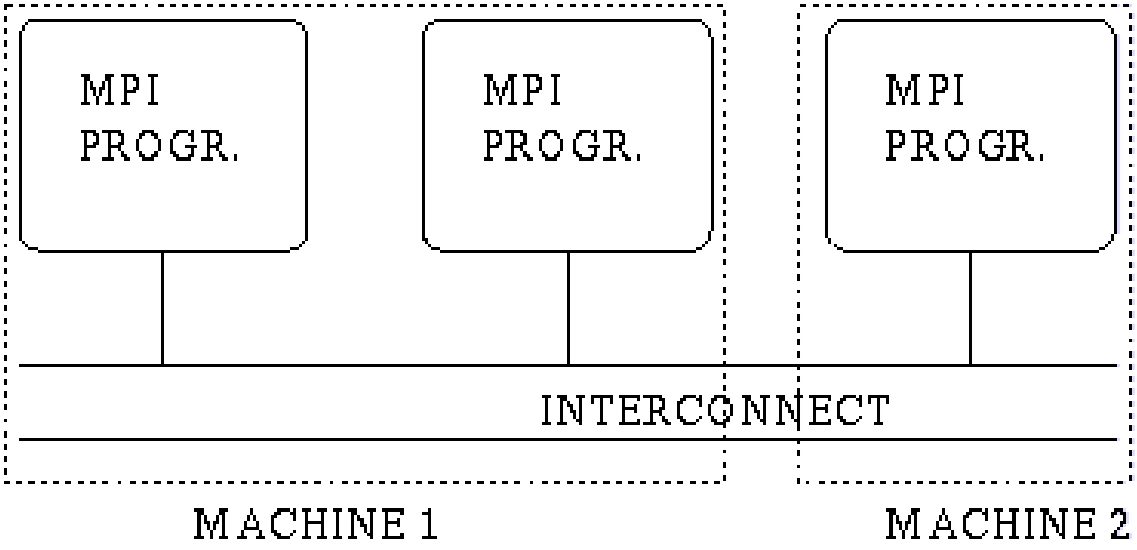
http://www.csm.ornl.gov/pvm/, last mod in 2011, inspected 2014



Parallel Virtual Machine (PVM)

- **Parallel Virtual Machine** (ORNL, CMU, and others)
- Target
  - heterogeneous networks (networks of workstations)
- Unix environment
  - task: Unix process, TCP/IP
- Dynamic configuration
  - add/delete hosts
- Architecture
  - PVM daemon on each host; task registration
  - Communication between named tasks
  - Design support for regular structures
  - No shared memory between tasks on same machine
  - no channels

http://www.open-mpi.org/, last mod feb 2015, inspected 2015



MPI: Message passing interface

- Parallel program equals virtual machine: one task (program) per processor in the machine
- Communication between named tasks (numbering from 0 on)
- Designed to be a general interface, not tied to a particular system
- No ``de facto'' standard, but result of experience and discussion
- In principle highly portable (at least a subset)
- No shared memory, no task creation, no channels

The Message-Passing Interface

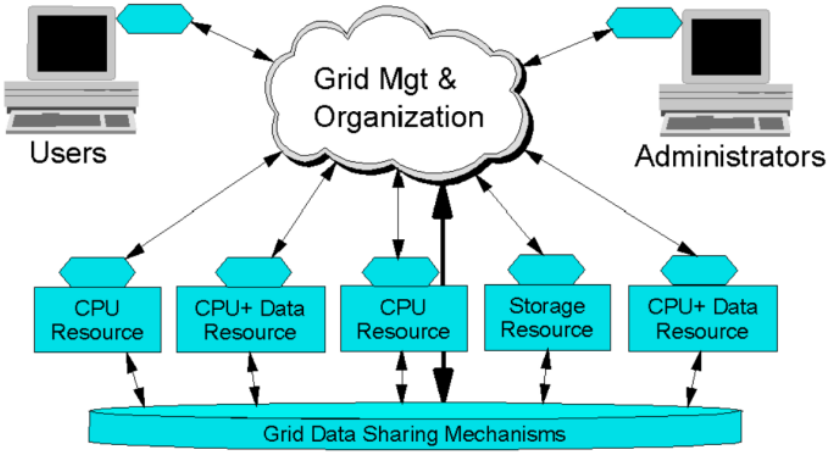
Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

MPI API

- *Task creation and grouping*: no new tasks; joining and leaving named groups; virtual topologies
- *Communication*: send/receive typed messages to named tasks; blocking and non-blocking communication; broadcasting; reduce (global sum),.....
- *Synchronization*: barrier; implicit in message passing
- *Actions to control resource allocation*: hardly
- *Basic i/o*: use local filesystems; standard output directed to "console"

Grids

- A Grid represents a *platform* for deploying MPI-like programs
  - a collection of machines
  - a management layer
    - scheduling
    - resource management
    - adding / removing machines
    - managerial domains
  - application handling support
    - dispatch
    - monitor
    - data management
- Example: the Globus Grid package



From Ferreira et.al. Introduction to Grid Computing with Globus

Evaluation: MPI, PVM

- Good
  - Wide acceptance and support, including supporting tools
  - Source-level portability of programs
  - Increasing OS support for MPI

- Not so good
  - Large overhead for task creation (PVM) or no new task creation at all (MPI)
  - Relatively large communication latencies (PVM)
  - Big! (large number of functions in order to cover "all possibilities")
    - MPI has functions supporting specific machines
  - No direct support for multi-tasking

- Example
  - cumbersome to design a task that performs a computation on its local data but also permits other tasks to inspect this data asynchronously

Remember

- What is the preferred abstract programming model?
  - the one that admits portable programs while not sacrificing too much performance
    - example: the Von Neumann machine has a 1-1 counterpart in imperative languages
  - the one that admits high programmer performance while not sacrificing too much machine performance

- Programming Model
  - explicit concurrency:
    - programmer control, e.g. support creation of independent tasks that communicate by passing messages
    - each such task may have internal concurrency with shared memory
    - this maps easily to machine models, and allows optimization
  - implicit concurrency:
    - Domain-specific languages with run-time support
    - this allows concurrency and failure transparencies and increases programmer performance