How to think about parallel programming

LIGO-G010232-00-Z

Programming Abstractions

- Programming involves thinking in abstractions
  - » Data abstractions: e.g., arrays
  - » Instruction abstractions: if/then/else, loops, case/switch
  - » Functions and subroutines (matrix-multiplication(), sin()) are further abstractions built upon abstractions
  - » Modularity
- Good abstractions simplify and organize the design space without restricting functionality
- The *von Neumann Machine*
  - » Abstraction that underlies modern computing
  - » Divides computers into hardware and software
  - » Divides hardware into a *central processing unit* and *memory*
  - » Divides software into *instructions* operating on *data*
  - » Not an exclusive or universal abstraction – e.g., LISP
- Parallel programming introduces a new layer of abstraction

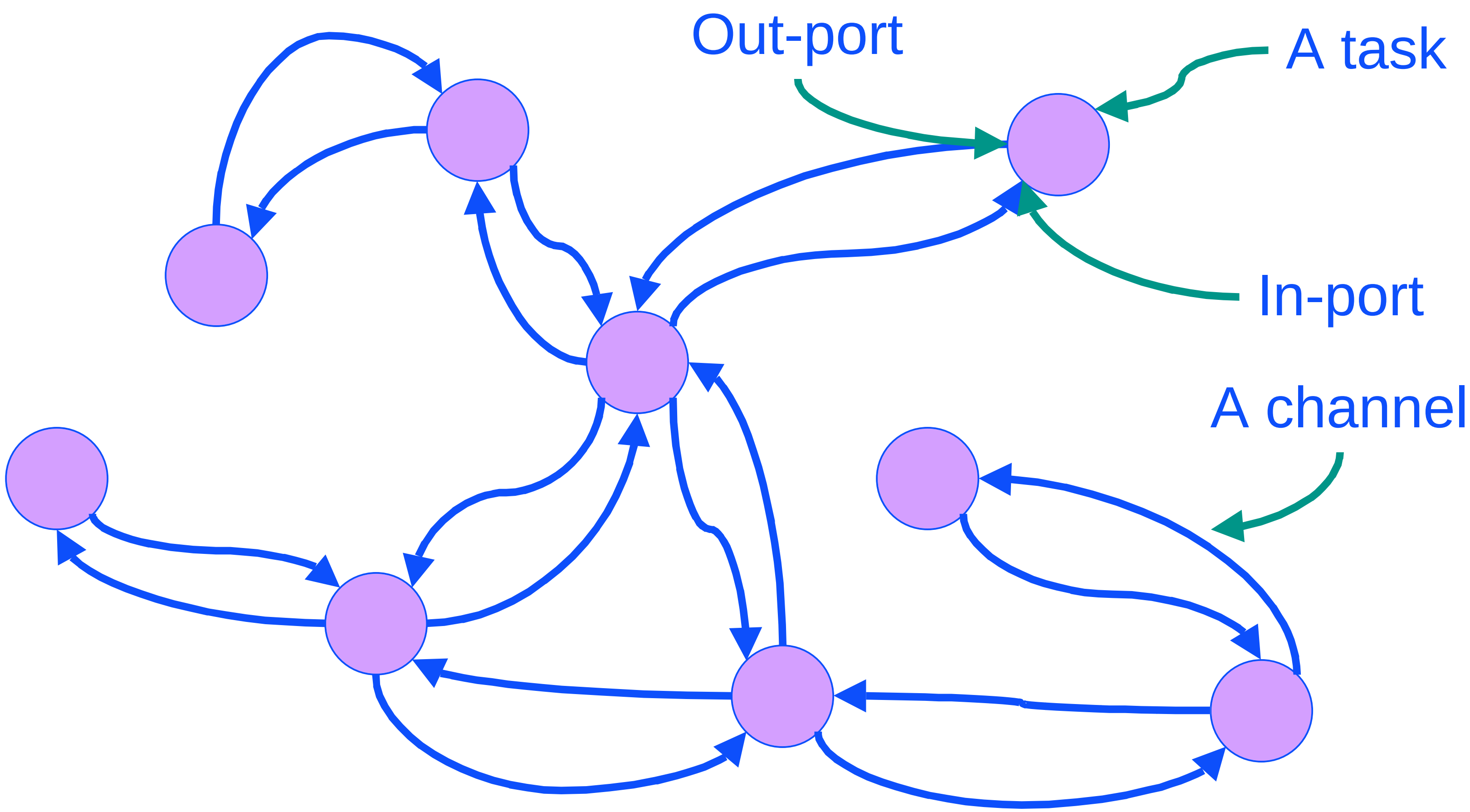
A Parallel Programming Model

- Focus: The Multicomputer
  - » A collection of von Neumann Machines that can communicate with each other via an interconnection network, or interconnect
  - » Node: an individual computer (mearning von Neumann machine)
  - » Implications
    - – Distributed memory: Each node has its own memory, whose contents are not directly accessible to other nodes
    - – Distributed instructions: Each node has its own cpu, executing its own set of instructions
  - » A Multiple Instructions, Multiple Data (MIMD) model
- Other models
  - » Single Instruction, Multiple Data (SIMD): One CPU (and set of instructions) acting (simultaneously) on many different data sets
  - » Multiple Instruction, Single Data (MISD): Many CPUs(and sets of instructions) acting (simultaneously) on the same data

Abstractions for Parallel Programming

- Each multicomputer node is a von Neumann machine
  - » All the usual abstractions apply to each node
- Two new abstractions – *task* and *channel* – deal with parallel computation on a multicomputer
  - » Task: a sequential program and its local memory (your basic computer), together with a set of in-ports and out-ports to communicate with other tasks.
  - » Channel: an in-port/out-port pair linking two tasks
- A parallel program consists of one or more tasks
  - » Tasks execute concurrently, and can be created and destroyed
- A task can perform four basic actions beyond reading and writing local memory
  - » Send messages on out-ports
  - » Receive messages on in-ports
  - » Create new tasks
  - » Terminate
- Send operations are asynchronous
- Receive operations are synchronous
- Channels can be created and destroyed, and references to channels can be included in messages
  - » Allows dynamical connectivity

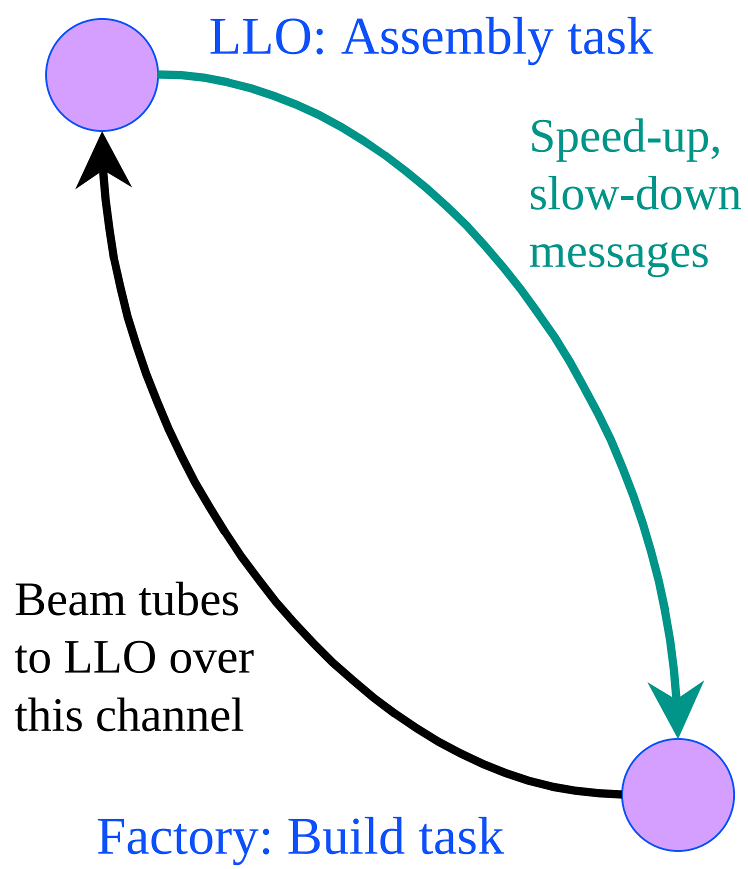
Tasks and Channels



LIGO-G010232-00-Z

Example: Building the LLO IFO arms

- Two tasks
  - » Building beam tubes at off-site factor
  - » Assembling beam tubes into an arm
  - » Each task executes own instructions concurrently
- Two communication channels
  - » From factory task to assembly task: “messages” are beam tubes
  - » From assembly task to factory task: “messages” are “send more beam tubes”, “stop sending beam tubes”



LIGO-G010232-00-Z

Tasks and Channels

- Tasks are not nodes
  - » Tasks are instructions and data; a node is a piece of hardware
  - » Multiple tasks can be mapped to a single node
  - » Single tasks can be mapped to multiple nodes
    - – This is subtle, and not commonly done, because task must be further organized so that instructions on each node are local with the data they operate on
- Tasks are the parallel computing equivalent of subroutines or functions
  - » Modularity in a parallel programming environment

Tasks, Data and Parallelism

- Tasks can do different things concurrently to the same data
  - » E.g., apply different templates or analysis methods to the same data
- Tasks can do the same things concurrently to different data
  - » E.g., apply the same templates to different data sets
- There are *many* ways to organize a parallel program
  - » No one "right" way
  - » Don't straight-jacket your thinking about parallelism

Message Passing

- LDAS uses the Message Passing programming model
  - » Tasks are identified by name
  - » Tasks interact by sending and receiving messages from other named tasks
- MPI: Message Passing Interface
  - » The programming language for message passing
- Six basic MPI "instructions":
  - » MPI_INIT: initiate an MPI computation
    - – *LDAS does this for you!*
    - – You will never do an MPI_INIT yourself
  - » **MPI_COMM_SIZE**: determine the number of processors available to you
  - » **MPI_SEND**: Send a message
  - » **MPI_COMM_RANK**: What node am I?
  - » **MPI_RECEIVE**: Receive a message
  - » MPI_FINALIZE: Terminate a MPI computation
    - – *LDAS does this for you!*
- Masters and Slaves
  - » Each MPI program has a single master task. All other tasks are slaves.
  - » The master is responsible for coordinating the action of the slaves