Parallel programming in Julia

Pierre Mabille

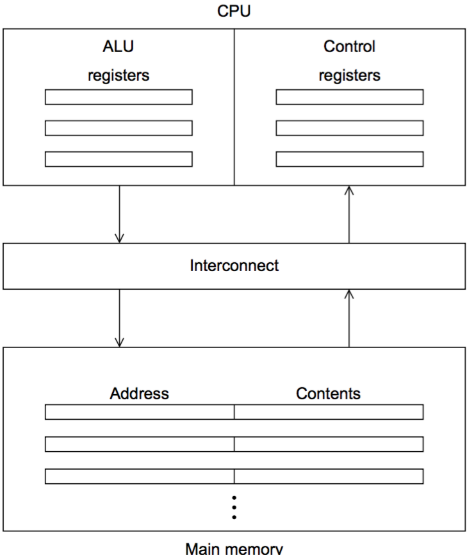
NYU Stern

April 22, 2015

Why parallel programming?

- Increasing density of transistors on integrated circuits generates increased power consumption and heat
  - simpler multicore processors, clusters (NYU/Stern HPC), rather than single complex processors
- Economic models with large state spaces, multiple constraints
- Global solutions

The Von-Neumann architecture



Parallel hardware

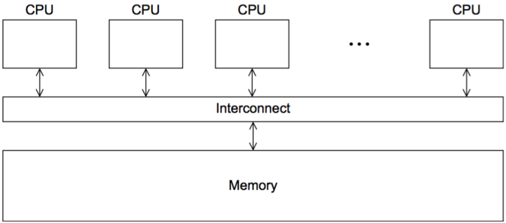
Flynn's taxonomy

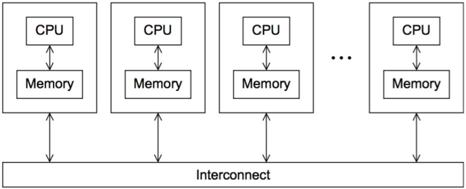
- Single instruction, single data (SISD)
- Single instruction, multiple data (SIMD)
- Multiple instruction, multiple data (MIMD)

Two types of MIMD

- Shared-memory system
  - one or more multicore processors
- Distributed-memory system (e.g. clusters of PCs)
  - nodes (PCs) are shared-memory
  - unified by grid

Shared vs. distributed memory





Julia's principles for parallel programming

Two performance factors: CPU speed, access to memory

Interface based on message-passing, implicit and one-sided

- you explicitly manage only one processor (the main one)

Key notions

- Remote reference: an object that can be used from any processor to refer to an object stored on a particular processor
- Remote call: a request by one processor to call a certain function on certain arguments on another (possibly the same) processor; returns a remote reference

Remote calls

Remote calls return immediately: the processor that made the call can then proceeds to its next operation while the remote call happens somewhere else

- Tasks run asynchronously on various processors

You can `wait` for a remote call on its remote reference, and you can obtain the full value of the result with `fetch`

To obtain a remotely-computed value immediately, use `remotecall_fetch(...)`, more efficient than `fetch(remotecall(...))`

Channels

Remote references always refer to an implementation of an `AbstractChannel`

`Channels` are fast means of inter-task communication

`Channel{T}(n::Int)` has maximum length `n` and holds objects of type `T`

Multiple readers can read off the channel via `fetch` and `take!`, multiple writers can add to the channel via `put!`

`isready` tests for the presence of any object in the channel

More commands at `http://docs.julialang.org/en/release-0.4/stdlib/parallel/`

More direct and simple: macros

`@spawn` and `@spawnat`

`@async` and `@sync`

`@everywhere`

`@parallel`

Code and packages availability

The code (functions, types...) must be available on any process that runs it

Same when loading codes or packages

- Use `@everywhere` to define an object on all processors
- `@everywhere begin ... end` for multiple objects

Parallel maps and loops

Parallel reduction: many iterations run independently over several processes, and then their results are combined using some function

`pmap`

- each function does a large amount of work

`@parallel for`

- each iteration is small

Reduction step implicit

`@parallel for` loops and shared arrays

Evaluation of `@parallel for` loops

- Iterations run on different processors and do not happen in a specified order
- Consequently, variables or arrays will not be globally visible
- Any variables used inside the parallel loop will be copied and broadcast to each processor
- Processors produce results which are made visible to the launching processor via the reduction

Variables in `@parallel for` loops

- Can include outside variables if read-only
- With a `SharedArray` each participating process has access to the entire array

Example: neoclassical growth model

Solve the model by VFI on one grid of various lengths (see notebook)

General lessons

- Parallel execution faster for large grids
- May be slower for small grids because of communication overhead
- Trade-off
- Understanding of general principles useful for debugging (e.g. `RemoteException` error)