Introduction to parallel and distributed computing in high-level programming languages: MATLAB and Julia

Henryk Modzelewski (hmodzelewski@eoas.ubc.ca)
&
Keegan Lensink (klensink@eoas.ubc.ca)
EOAS SLIM/WFRT teams

WestGrid Research Computing Summer School - UBC - June 2017

Hopeful agenda

- Some preparation - WestGrid access and setup

- Why high-level languages?
- Why Parallel & Distributed?
- Basics: MATLAB vs. Julia
- Parallel/Distributed features: MATLAB vs. Julia
- Debugging and profiling
- Extensions via other languages: MATLAB vs. Julia
- Some simple examples and exercises

WestGrid access & setup

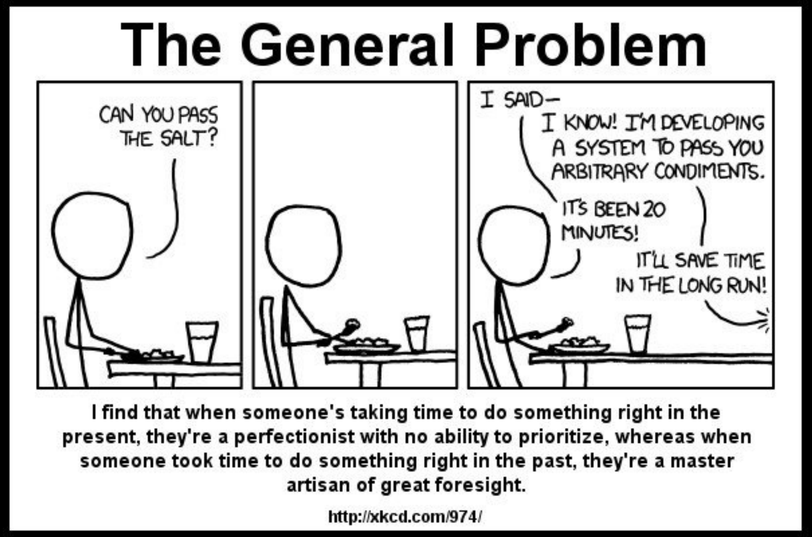
- Host: $ ssh -Y user_name@grex.westgrid.ca
- Get resources:
  - $ qsub -I -l nodes=1:ppn=3;walltime=3:00:00
- Environment modules:
  - $ module load python/2.7.9-gcc52
  - $ module load julia/0.5.2-bin
- $ export JULIA_LOAD_PATH=.

WestGrid access & setup

- Clone GIT repo
  - https://github.com/henryk-modzelewski/MvsJ.git
- Add packages (in $ julia)
  - Pkg.add(“BenchmarkTools”)
  - Pkg.add(“Gallium”)
  - Plg.add(“DistributedArrays”)

Why high-level languages?

Why high-level languages?



Pareto principle
80/20 (?)

- To develop faster
- To build less code
  - utilize libraries build by others
- To debug and profile easier
- Natively domain-specific languages (BLA)
  - or easy to adapt (OOP)
  - or having supporting packages

- To run slower :) or maybe not.

Propaganda!!

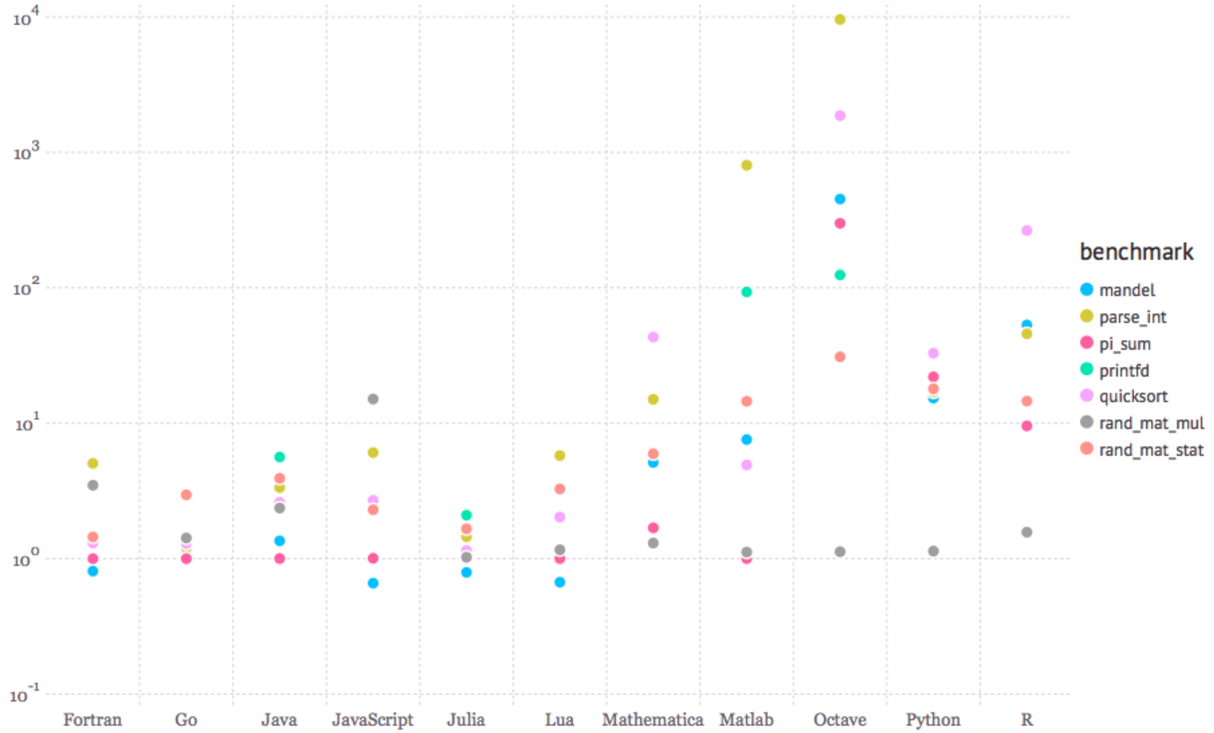


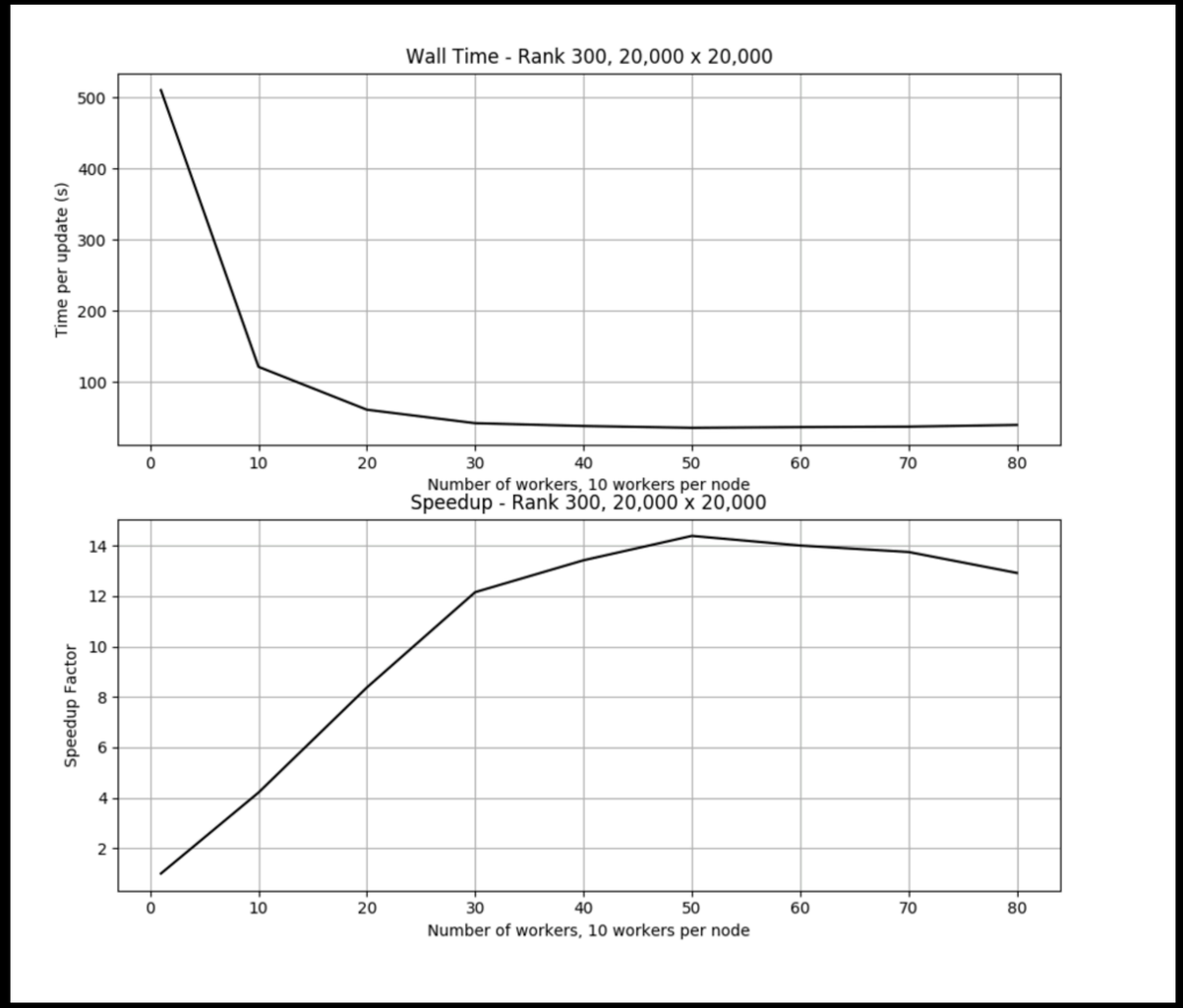
Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

C and Fortran compiled with gcc 5.1.1. C timing is the best timing from all optimization levels (-O0 through -O3). C, Fortran and Julia use OpenBLAS v0.2.14. The Python implementations of `rand_mat_stat` and `rand_mat_mul` use NumPy (v1.9.2) functions; the rest are pure Python implementations. Plot created with Gadfly and IJulia from this notebook.

Why parallel and distributed?

- To run applications faster
- To process more data
- To run bigger problems

- **Above subject to potential for scaling**
- Serial vs parallel portion of the code
- Miracles do not happen
  - they require hard work
  - or just do not happen :)



Basics - Big picture

MATLAB vs Julia - big picture

	MATLAB	Julia
License	Proprietary	Open Source
Cost	License	Free
Libraries	Core licensed + free	Core free + licensed
Documentation	In application & on-line	In application & on-line
Installation	Requires admin	User installable

MATLAB vs Julia - big picture

	MATLAB	Julia
Robustness	Old foe minor changes	New friend changes quite a lot
Scalability	Supported up to 256*	Supported yet unknown limits
Runtime	Interpreter	JiT compiler
OOP	Supported via classes single-dispatch	Supported via types multiple-dispatch
Garbage collection	Automatic	Automatic

MATLAB vs Julia - big picture

	MATLAB	Julia
Profiling	Easy and thorough	Basic profiling tools
Debugging	Easy and flexible GUI support	Still quite crude CLI only

Resources

- MATLAB
  - https://www.mathworks.com/help/matlab/
  - http://www.mathworks.com/matlabcentral/
  - https://www.youtube.com/user/MATLAB

- Julia
  - https://julialang.org
  - https://en.wikibooks.org/wiki/Introducing_Julia
  - https://www.youtube.com/user/JuliaLanguage
  - https://github.com/JuliaLang

Basics - running front-end

Basics - running

- In workspace
  - use command:
    - $ matlab [-nodesktop | -display]
  - execute from there
  - type function name for external script
  - finish with exit

- From shell
  - use command:
    - $ matlab [-nodesktop | -display] -r ‘script/function_name*[; exit]*’

Basics - running

- In workspace
  - use command:
    - $ julia
  - execute from there
  - use include() for script
  - finish with ^D or quit()

- From shell
  - use command:
    - $ julia script_name
    - $ julia -L lib_name scrip_name
  - shebang and execute perms
    - #!/usr/bin/env julia
    - $ chmod u+x script_name

Using workspace

	MATLAB	Julia (REPL)
Line suppression	…;	…;
SHELL escape	!	;
Help	help	?
Search		apropos
Locate	which	
Reset	clear […]	workspace()

Basics - language fundamentals

Basics - assignments and operators

- Both languages support the typical syntax used in other languages
- Assignments look the same
- Similar operators
- Standard operator precedence
  - same precedent asses left-to-right
- Function calls look the same

```
a=1

b=a+1

c=a*b

x=foo(y,a,b,c)
```

Build-in simple types

- Numerals:
  - integers with different precision
  - floats/complex with single/double precision
  - hard to predict/control the outcome
  - computations are cared in single or double precision (!)
    - typically downgrades to single in binary operators (!)
- Strings (characters are size (1,1) strings)
- Date & Time
- …

Build-in simple types

- Numerals:
  - integers with different precision
  - floats/complex with different precision
  - infers result of operation from higher-precision components
    - promote_type()
- Strings
- Characters (null size)
- Date & Time
- …

Built-in complex types

- Numerical arrays column-major, indexed (i,…)
  - (either double or single)
  - support slicing with ‘:’ e.g. (:,i:j)
- Cell arrays (single dim) - hold anything, indexed {i}
  - slicing {i:j} returns separate elements
- Structures - expandable
- Composite() - for parallel computations

https://www.mathworks.com/help/matlab/data-types_data-types.html

Built-in complex types

- Numerical arrays column-major, indexed [i,…]
  - Matrix and Vector are different types of Array
- “Any” arrays - anything
- Tuples, immutable, indexed with [i]
  - above support slicing with ‘:’, e.g. [i:j,:]

https://docs.julialang.org/en/latest/manual/types.html

Basics - some notable differences

Basics - some notable differences

- numeral assignments always create type ‘double’
- complex using ‘j’
- scalars/charecters have size (1,1)
- boolean (true/false) negate with ‘~’
- strings surrounded by ‘double quote’
- characters surrounded by ‘single quote’ (!)
- strings are arrays of characters
- string concatenation via function strcat
- comments start with ‘%’

```
a=1
whos a
b=1.
whos b

d=1+j*4

size(a)

T=true
F=~T
```

```
s=‘hello’
c=s(1)

ab=strcat(‘a','b')
ab=[‘a’,’b']
```

Basics - some notable differences

- numeral assignments default integer/flaot/complex types, other type require convert()
- complex using ‘im’
- scalars/characters have null size
- boolean (true/false) negate with ‘!’
  - strings surrounded by “double quote”
  - characters surrounded by ‘single quote’
  - strings are arrays of characters
  - strings concatenate via ‘*’
  - strings can be interpolated
  - in-place functions - names end with ‘!’ convention
  - +=. -=, *=, /=
  - comments start with ‘#’

```
a=1
typeof(a)
b=1.
typeof(b)

d=1+im*4

size(a)


T=true
F=!true
```

```
S=“test”
C=‘c”

S2=“test1”*”test2”
S3=“This is $S2”
S4=“This is $a”

A=[1, 2]
push!(A,3)
```

Basics - flow control

Basics - flow control

- for loops
  - for i=1:10; …; end
  - for i=1:2:13; …; end
    - continue & break
- while *cond*; …; end
  - continue
- switch *EXP*; case *exp* …; otherwise …; end
  - no break
- if *cond* … ; elseif *cond2* …; else… ; end
- exceptions
  - throw() & rethrow()
  - try …; catch …; end

Basics - flow control

- for loops
  - for i=1:10
  - for i=1:2:11
  - for e in ‘iterable’
    - continue & break
- while *cond*; …; end
  - continue & break
- if *cond* … ; elseif *cond2* …; else… ; end
  - if *cond* … end
  - ternary operator: x = *cond* ? … : …
- exceptions
  - throw() & rethrow()
  - try …; catch; end

Basics - function

Functions

- One accessible function per file
- Arguments
  - required (positional)
  - optional (positional)
  - keywords ‘key’,’value’ sequence
- No type matching
  - processing arguments in order
  - or use built-in parser
- In-place functions
- Anonymous functions
- Function pointers with ‘@‘

```
% my function file

%main
function [x,y]=foo(a,b,c,’key’,’value’)
  …
end

%private function
function c=extra(d,e)
  …
end

% private in-place
function c=inplace(c)
  …
end
```

```
function op = opCurvelet(m, n, nbscales, nbangles,…
                    finest,ttype, is_real)

  assert( isscalar(m) && isscalar(n),['Please ensure'...
          ' sizes are scalar values']);
  if nargin < 3, nbscales = max(1,ceil(log2(min(m,n)) - 3)); end;
  if nargin < 4, nbangles = 16;                              end;
  if nargin < 5, finest = 0;                                 end;
  if nargin < 6, ttype = 'WRAP';                             end;
  if nargin < 7, is_real = 1;                                end;
  assert( strcmp(ttype,'WRAP') || strcmp(ttype,'ME'),...
              ['Please ensure ttype is set correctly. Options are'...
              ' "WRAP" for a wrapping transform and "ME" for a'...
              ' mirror-extended transform']);
  assert( isscalar(nbscales) && isscalar(nbangles),...
              'Please ensure nbscales and nbangles are scalar values');
  assert((any(finest == [0 1 2])) && (is_real==0||is_real==1),...
              'Please ensure finest and is_real are appropriate values');
  if finest==0, assert( nbscales>1, ['Please ensure that '...
                  'm and n are large enough for nbscales to be '...
                  'greater than 1 while finest is set to 0']);
  end
…
```

```
function x = poMatCon(pathname,varargin) % Constructor for poMatCon

   % Parse param-value pairs using input parser
   p = inputParser;
   p.addParamValue('precision','double',@ischar);
    p.addParamValue('repeat',0,@isscalar);
    p.addParamValue('readonly',0,@isscalar);
    p.addParamValue('distribute',0,@ isscalar);
    p.addParamValue('copy',0,@isscalar);
    p.KeepUnmatched = true;
    p.parse(varargin{:});

    if (isdir(pathname)) % Loading file
       if (p.Results.copy == 0) % overwrite case
          headerIn = SDCpckg.Reg.io.NativeBin.serial.HeaderRead(pathname);
          td = pathname;
       else % no overwrite
           td = SDCpckg.Reg.io.makeDir();
           SDCpckg.Reg.io.NativeBin.serial.FileCopy(pathname,td);
           headerIn = SDCpckg.Reg.io.NativeBin.serial.HeaderRead(td);
       end
    else
…
```

Functions

- No limit on # of functions in file
- From week to strong typing
- Arguments
  - required (positional)
  - optional (positional until done)
  - keywords ‘key’=’value’ after ‘;’
  - type matching and all above
    - still a bit shaky
- Returns last-line result or via return statement
- Anonymous functions
- Functions are objects

```
function foo(a,b,c=1,d=7;verb=true)
  …
  return res
end

function extra(d::Int,e::Float64;verb::Bool=true)
  …
  res = something
end

function extra{DT}(d::DT,e::DT)::DT
  …
  res
end
```

```
function joCoreBlock(ops::joAbstractLinearOperator...;kwargs...)
  isempty(ops) && throw(joCoreBlockException("empty argument list"))
   l=length(ops)
   for i=1:l
       deltype(ops[i])==deltype(ops[1]) || throw(joCoreBlockException("domain type mismatch for $i operator"))
       reltype(ops[i])==reltype(ops[1]) || throw(joCoreBlockException("range type mismatch for $i operator"))
   end
   mykws=Dict(kwargs[i][1]=>kwargs[i][2] for i in 1:length(kwargs))
   mo=Base.deepcopy(get(mykws, :moffsets, zeros(Int,0)))
      typeof(mo)<:AbstractVector || throw(joCoreBlockException("moffsets must be a vector"))
      eltype(mo)<:Integer || throw(joCoreBlockException("moffsets vector must have integer elements"))
      (length(mo)==l || length(mo)==0) || throw(joCoreBlockException("lenght of moffsets vector does not match number of operators"))
   no=Base.deepcopy(get(mykws, :noffsets, zeros(Int,0)))
      typeof(no)<:AbstractVector || throw(joCoreBlockException("noffsets must be a vector"))
      eltype(no)<:Integer || throw(joCoreBlockException("noffsets vector must have integer elements"))
      (length(no)==l || length(no)==0) || throw(joCoreBlockException("lenght of noffsets vector does not match number of operators"))
   ws=Base.deepcopy(get(mykws, :weights, zeros(0)))
      typeof(ws)<:AbstractVector || throw(joCoreBlockException("weights must be a vector"))
      (length(ws)==l || length(ws)==0) || throw(joCoreBlockException("lenght of weights vector does not match number of operators"))
   name=get(mykws, :name, "joCoreBlock")
      typeof(name)<:String || throw(joCoreBlockException("name must be a string"))
   ME=get(mykws, :ME, 0)
      typeof(ME)<:Integer || throw(joCoreBlockException("ME must be Integer"))
       ME>=0 || throw(joCoreBlockException("ME must be >=0"))
   NE=get(mykws, :NE, 0)
       typeof(NE)<:Integer || throw(joCoreBlockException("NE must be Integer"))
       NE>=0 || throw(joCoreBlockException("NE must be >=0”))
…
```

Functions

- From week to strong typing
- Parametric types
- Type stability
  - ensuring that compiler has least work to guess types
  - types assessed on the function boundary
- **Type stability is paramount for performance**
  - @code_warntype

```
function jo_convert{VT<:Integer}(DT::DataType,vin::AbstractArray{VT},warning::Bool=true)
    DT==VT && return vin
   if DT<:Integer
        if typemax(DT)>typemax(VT)
            vout=convert(AbstractArray{DT},vin)
        else
            throw(joUtilsException("jo_convert: Refused conversion from $VT to $DT."))
        end
    else
        vout=convert(AbstractArray{DT},vin)
    end
    return vout
end

function jo_convert{VT<:AbstractFloat}(DT::DataType,vin::AbstractArray{VT},warning::Bool=true)
    DT==VT && return vin
   if !(DT<:Integer)
        vout=convert(AbstractArray{DT},vin)
    else
        throw(joUtilsException("jo_convert: Refused conversion from $VT to $DT."))
    end
    return vout
end

function jo_convert{VT<:Complex}(DT::DataType,vin::AbstractArray{VT},warning::Bool=true)
    DT==VT && return vin
   if DT<:Complex
        vout=convert(AbstractArray{DT},vin)
    elseif DT<:AbstractFloat
        (warning && jo_convert_warn) && warn("jo_convert: Inexact conversion from $VT to $DT. Dropping imaginary part.")
        vout=convert(AbstractArray{DT},real(vin))
    else
        throw(joUtilsException("jo_convert: Refused conversion from $VT to $DT."))
    end
    return vout
end
```

Demonstration: in directory Loop

Exercise: in directory Clip

Use clip.m to produce Julia version in myclip.jl
- watch for type stability

Basics - OOP

Basics - OOP

- Inheritance, encapsulation, and polymorphism
- Classes
  - properties (public, protected, and private)
  - methods (public, protected, and private)
  - value and handle classes
- Single-dispatch - methods bound to class object

Basics - OOP

- Subtyping (type tree and associations) and ad hoc polymorphism (function overloading)
- Types
  - mutable or immutable
  - constructors (outer and inner)
  - abstract types
  - type aliases
- Methods (functions)
- Multiple-dispatch - methods bound by function's signature

Basics - OOP

- Modules
  - full-protection of module types and variables
  - semi-protection of methods (overloading for different types)

Real code (domain-specific language):

https://github.com/slimgroup/JOLI.jl.git

Demonstration: in directory OOP

Exercise: in directory OOP

Use figures.jl

1. addSquare
2. add >, >=,<=,==

Adding and extending code

Adding 3-rd party code

- No package manager
  - Typically distributed as tar or zip archives
  - Installs anywhere
  - Just unzip/un-tar the archive somewhere and
    - add location to the path (next slides)

- Typical places:
  - ~/matlab
  - ~/Documents/MATLAB

Adding 3-rd party code

- Package Manager
  - GIT repositories
  - Installs into ~/.julia/v#.#/pckg_name
  - using & import

- zip/tar archives
  - Just unzip/un-tar the archive somewhere and
    - add location to the path (next slides)

Adding 3-rd party code

- Registered packages
  - Pkg.add(“pckg_name”)
- Unregistered packages from GIT repositories
  - Pkg.clone(“pckg_url”[,local_name])
- Checking out branches
  - Pkg.checkout(“pckg_name”,”branch_name”)

Organizing your source code

- Functions
- Classes
- Packages
  - SHELL environment MATLABPATH
  - addpath("/home/me/Matlab"*[,'-begin'|'-end']*)
  - addpath /home/me/Matlab *[-begin|-end]*
- **MATLAB will find everything on the path**

Organizing your source code - Packages

Directory listing

+SLIM_APPS

+SLIM_APPS/available.m

+SLIM_APPS/+tools

+SLIM_APPS/+tools/Miscellaneous.m

+SLIM_APPS/+tools/+algorithms

+SLIM_APPS/+tools/+algorithms/REPSI.m

+SLIM_APPS/+tools/+algorithms/TimeModeling.m

+SLIM_APPS/+tools/+utilities

+SLIM_APPS/+tools/+utilities/pSPOT.m

+SLIM_APPS/+tools/+utilities/SPOT_SLIM.m

Packages create namespaces

```
>> SLIM_APPS.available

>> SLIM_APPS.tools.Miscellaneous(…)

>> SLIM_APPS.tools.utilities.pSPOT(…)
```

Organizing your source code

- Modules
  - SHELL environment JULIA_LOAD_PATH
  - append: push!(LOAD_PATH,“/home/me/julia”)
  - prepend: unshift!(LOAD_PATH,“/home/me/julia")
- **using & import will find all modules on the path**

- Functions (not recommended)
  - include() & require() & reload()
  - include mostly for organizing modules (relative or full path)
  - **require() & reload() will find functions on the path**

Parallel/Distributed features

Scalability - vertical vs. horizontal scaling

- vertical
  - expanding resources on single computing server
  - shared memory
- horizontal
  - adding more computing servers
  - distributing memory

Scalability - strong vs. weak scaling

- strong
  - expanding resources for same problem size
  - decreasing problem size per process
  - (sub-size=size/workers)
- weak
  - increase problem proportional to increased resource
  - constant problem size per process
  - (size=sub-size*workers)

Parallel tools

- No explicit multi-threading
- Communication
  - Master-worker model from main client
  - MPI mode between workers
- parfor loop (no nesting)
- spmd block (no nesting)
- distributed/codistributed arrays
  - distributed array constrictors
  - codistributed array builders

Parallel tools

- Functions:
  - parpool (interactive)
  - batch (non-interactive)
    - script submission
    - function call

Parallel tools

- Parallel invocation via functions:
  - parpool(#) - interactive
  - batch - noninteractive jobs
    - can also execute function remotely

Parallel tools

- Multi-threading
  - Implicit for some native components
  - Threads module (experimental)
    - Threads.@threads
- Communication between workers - RPC
  - “Master”-workers model (actor?)
- Global parallel calls
  - @parallel for (SharedArrays)
  - pmap

Parallel tools

- Feature - response to
  - @spawn & @spawnat
  - remotecall() - non-blocking
  - remotecall_wait() - blocking
  - remotecall_fetch() - blocking
  - wait() & fetch()

Parallel tools

- RemoteChannel - rewritable communication path
- @sync & @async for more sophisticate control
- Distributed arrays - Package DistributedArrays
  - spmd functionality in next v0.6 version of Julia

Parallel tools

- Thread control:
  - set threads: $ export JULIA_NUM_THREADS=#
  - BLA operators: > BLAS.set_num_threads(#)
- Parallel invocation:
  - command line: $ julia -p # [--machinefile <file>]
  - addprocs() and rmprocs()

Example; parallel command execution
in CodeExamples

Debugging and profiling - Keegan Lensink

Extensions via other languages: MATLAB vs. Julia

- MATLAB coder - compiling native MATLAB code
- Java - MATLAB is partly Java, and supports Java classes
- C/Fotran API - a bit cumbersome to use
  - mex extensions - great for
    - accelerating
    - OpenMP/multithreading

Extensions via other languages: MATLAB vs. Julia

- Python - Package PyCall
- MATLAB - Package MATLAB
- C/Fotran via ccall (dlopen)
  - great for
    - accelerating
    - OpenMP/multithreading

Advection examples - code analysis

	MATLAB	Julia
Serial	advection.m	advection.jl
Threaded		advectionT.jl
parfor @parallel	advectionPF.m advectionPFV.m	advectionPS.jl advectionPSS.jl
pmap		advectionPMAP.jl
spmd	advectionD.m	

The End