Parallel Ccomputing with Julia

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS2101

Plan

1. Preliminaries: Coroutines
2. Julia's Prnciples for Parallel Computing
3. Tips on Moving Code and Data
4. Around the Parallel Julia Code for Fibonacci
5. Parallel Maps and Reductions
6. Synchronization
7. Distributed Arrays
8. A Simple Simulation Using Distributed Arrays

Plan

Tasks (aka Coroutines)

Tasks

- Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner
- This feature is sometimes called by other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations.
- When a piece of computing work (in practice, executing a particular function) is designated as a Task, it becomes possible to interrupt it by switching to another Task.
- The original Task can later be resumed, at which point it will pick up right where it left off

Producer-consumer scheme

The producer-consumer scheme

- One complex procedure is generating values and another complex procedure is consuming them.
- The consumer cannot simply call a producer function to get a value, because the producer may have more values to generate and so might not yet be ready to return.
- With tasks, the producer and consumer can both run as long as they need to, passing values back and forth as necessary.
- Julia provides the functions `produce` and `consume` for implementing this scheme.

Producer-consumer scheme example

```
function producer()
  produce("start")
  for n=1:2
    produce(2n)
  end
  produce("stop")
end
```

To consume values, first the producer is wrapped in a Task, then consume is called repeatedly on that object:

```
ulia> p = Task(producer)
Task

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
"stop"
```

Tasks as iterators

A Task can be used as an iterable object in a for loop, in which case the loop variable takes on all the produced values:

```
julia> for x in Task(producer)
         println(x)
       end
start
2
4
stop
```

Plan

Julia's message passing principle

Julia's message passing

- Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processors in shared or distributed memory.
- Julias implementation of message passing is one-sided:
  - the programmer needs to explicitly manage only one processor in a two-processor operation
  - these operations typically do not look like message send and message receive but rather resemble higher-level operations like calls to user functions.

Remote references and remote calls

Two key notions: remote references and remote calls

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

How remote calls are handled in the program flow

- Remote calls return immediately: the processor that made the call can then proceeds to its next operation while the remote call happens somewhere else.
- You can wait for a remote call to finish by calling wait on its remote reference, and you can obtain the full value of the result using fetch.

Remote references and remote calls: example

```
$ ./julia -p 2

julia> r = remote_call(2, rand, 2, 2)
RemoteRef(2,1,5)

julia> fetch(r)
2x2 Float64 Array:
 0.60401   0.501111
 0.174572  0.157411

julia> s = @spawnat 2 1+fetch(r)
RemoteRef(2,1,7)

julia> fetch(s)
2x2 Float64 Array:
 1.60401  1.50111
 1.17457  1.15741
```

Commnets on the example

- Starting with `julia -p n` provides n processors on the local machine.
- The first argument to `remote_call` is the index of the processor that will do the work.
- The first line we asked processor 2 to construct a 2-by-2 random matrix, and in the third line we asked it to add 1 to it.
- The `@spawnat` macro evaluates the expression in the second argument on the processor specified by the first argument.

More on remote references

```
julia> remote_call_fetch(2, ref, r, 1, 1)
0.10824216411304866
```

`remote_call_fetch`

- Occasionally you might want a remotely-computed value immediately.
- The function `remote_call_fetch` exists for this purpose.
- It is equivalent to `fetch(remote_call(...))` but is more efficient.
- Note that `ref(r,1,1)` is equivalent to `r[1,1]`, so this call fetches the first element of the remote reference r.

The macro `@spawn`

The macro `@spawn`

- The syntax of `remote_call` is not especially convenient.
- The macro `@spawn` makes things easier:
  - It operates on an expression rather than a function, and
  - chooses the processor where to do the operation for you

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1+fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

Remarks on the example

- Note that we used `1+fetch(r)` instead of `1+r`. This is because we do not know where the code will run, so in general a `fetch` might be required to move `r` to the processor doing the addition.
- In this case, `@spawn` is smart enough to perform the computation on the processor that owns `r`, so the fetch will be a no-op.

Plan

Availability of a function to processors (1/2)

One important point is that your code must be available on any processor that runs it. For example, type the following into the julia prompt

```
julia> function rand2(dims...)
         return 2*rand(dims...)
       end

julia> rand2(2,2)
2x2 Float64 Array:
 0.153756  0.368514
 1.15119   0.918912

julia> @spawn rand2(2,2)
RemoteRef(1,1,1)

julia> @spawn rand2(2,2)
RemoteRef(2,1,2)

julia> exception on 2: in anonymous: rand2 not defined
```

Availability of a function to processors (2/2)

In the previous example, Processor 1 knew about the function rand2, but processor 2 did not. To make your code available to all processors, the `require` function will automatically load a source file on all currently available processors:

```
julia> require("myfile")
```

In a cluster, the contents of the file (and any files loaded recursively) will be sent over the network.

Data Movement (1/4)

Motivation

- Sending messages and moving data constitute most of the overhead in a parallel program.
- Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability.
- To this end, it is important to understand the data movement performed by Julias various parallel programming constructs.

Data Movement (2/4)

`fetch` and `@spawn`

- `fetch` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine.
- `@spawn` (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation.
- Consider these two approaches to constructing and squaring a random matrix
- Which one is the most efficient?

```
# method 1
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
fetch(Bref)
```

Data Movement (3/4)

```
# method 1
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
fetch(Bref)
```

Answer to the question

- The difference seems trivial, but in fact is quite significant due to the behavior of `@spawn`.
- In the first method, a random matrix is constructed locally, then sent to another processor where it is squared.
- In the second method, a random matrix is both constructed and squared on another processor.
- Therefore the second method sends much less data than the first.

Data Movement (4/4)

Remarks on the previous example

- In the previous toy example, the two methods are easy to distinguish and choose from.
- However, in a real program designing data movement might require more thought and very likely some measurement.
- For example, if the first processor needs matrix `A` then the first method might be better.
- Or, if processing `A` is expensive but only the current processor has it, then moving it to another processor might be unavoidable.
- Or, if the current processor has very little to do between the `@spawn` and `fetch(Bref)` then it might be better to eliminate the parallelism altogether.
- Or imagine `rand(1000,1000)` is replaced with a more expensive operation. Then it might make sense to add another `@spawn` statement just for this step.

Plan

Fibonacci (1/4)

```
  _       _ _(_)_     |  A fresh approach to technical computing
 (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
  _ _   _| |_  __ _   |  Type "help()" to list help topics
 | | | | | | |/ _` |  |
 | | |_| | | | (_| |  |  Version 0.2.0-prerelease+3622
_/ |\__'_|_|_|\__'_|  |  Commit c9bb96c 2013-09-04 15:34:41 UTC
|__/                   |  x86_64-redhat-linux

ulia> addprocs(3)
3-element Array{Any,1}:
 2
 3
 4

julia> @everywhere function fib(n)
                  if (n < 2) then
                      return n
                  else return fib(n-1) + fib(n-2)
                  end
              end
```

Fibonacci (2/4)

```
julia> z = @spawn fib(10)
RemoteRef(3,1,8)

julia> fetch(z)
55

julia> @time [fib(i) for i=1:45];
elapsed time: 32.241445075 seconds (416 bytes allocated)
```

Fibonacci (3/4)

```
julia> @everywhere function fib_parallel(n)
           if (n < 40) then
               return fib(n)
           else
               x = @spawn fib_parallel(n-1)
               y = fib_parallel(n-2)
               return fetch(x) + y
           end
       end

julia>

julia> @time [fib_parallel(i) for i=1:45];
elapsed time: 14.295663044 seconds (655820 bytes allocated)
```

Fibonacci (4/4)

```
julia> @time @parallel [fib(45+i) for i=1:4]
elapsed time: 21.654079705 seconds (26927736 bytes allocated)
4-element DArray{Int64,1,Array{Int64,1}}:

 1836311903
 2971215073
 4807526976
 7778742049

julia> @time [fib(45+i) for i=1:4];
elapsed time: 188.761594844 seconds (80 bytes allocated)
```

Plan

A first example of parallel reduction

```
julia> @everywhere function count_heads(n)
           c::Int = 0
           for i=1:n
               c += randbool()
           end
           c
       end

julia> a = @spawn count_heads(100000000)
RemoteRef(7,1,31)

julia> b = @spawn count_heads(100000000)
RemoteRef(2,1,32)

julia> fetch(a)+fetch(b)
99993168
```

- This simple example demonstrates a powerful and often-used parallel programming pattern: *reductuon*.
- Many iterations run independently over several processors, and then their results are combined using some function.

Parallel reduction using `@parallel` (1/4)

Usage of parallel for loops

- In the previous example, we use two explicit `@spawn` statements, which limits the parallelism to two processors.
- To run on any number of processors, we can use a parallel for loop, which can be written in Julia like this:

```
nheads = @parallel (+) for i=1:200000000
  randbool()
end
```

Comments

- This construct implements the pattern of
  - assigning iterations to multiple processors, and
  - combining them with a specified reduction (in this case (+)).
- Notice that the reduction operator can be omitted if it is not needed
- However, the semantics of such a parallel for-loop can be dramatically different from its serial elision. As we shall see on the example of the next slide.

Parallel reduction using `@parallel` (2/4)

```
julia> a = zeros(4)
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0

julia> @parallel for i=1:4
              a[i] = i
              end

julia> a
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0

julia> for i=1:4
          a[i] = i
       end

julia> a
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
```

Parallel reduction using `@parallel` (3/4)

Evaluation of a `@parallel` for-loop

- Iterations run on different processors and do not happen in a specified order,
- Conseqnently, 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 lauching processor via the reduction.
- This explains why the following code will not work as intended:

```
julia> @parallel for i=1:4
                a[i] = i
                end
```

Comments on the example

- Each processor will have a separate copy if it.
- Parallel for loops like these must be avoided

Parallel reduction using `@parallel` (4/4)

Use of "outside" variables in `@parallel` for-loops

- Using outside variables in parallel loops is perfectly reasonable if the variables are read-only. See the example on the next slide.
- In some cases no reduction operator is needed, and we merely wish to apply a function to all elements in some collection.
- This is another useful operation called parallel *map*, implemented in `Julia` as the `pmap` function.
- For example, we could compute the rank of several large random matrices in parallel as follows:

```
julia> M = [rand(1000,1000) for i=1:4];

julia>

julia> pmap(rank, M)
4-element Array{Any,1}:
 1000
 1000
 1000
 1000
```

Use of "outside" variables in `@parallel` for-loops

```
julia> toc()
elapsed time: 0.026080394 seconds
0.026080394

julia> M = [rand(1000,1000) for i=1:4];

julia> tic()
0x000a911d4bdfa458

julia> R = [@spawnat i rank(M[i]) for i=1:4]
4-element Array{Any,1}:
 RemoteRef(1,1,180)
 RemoteRef(2,1,181)
 RemoteRef(3,1,182)
 RemoteRef(4,1,183)

julia> S = 0
0

julia> for i=1:4
          S = S + fetch(R[i])
       end

julia> S
4000

julia> toc()
elapsed time: 1.436392165 seconds
1.436392165

julia> @time @parallel (+) for i=1:4
                rank(M[i])
       end
elapsed time: 1.018056042 seconds (235219708 bytes allocated)
4000
```

Plan

How does Julia's schedule computations?

Julia's scheduling strategy is based on tasks

- Julias parallel programming platform uses Tasks (aka Coroutines) to switch among multiple computations.
- Whenever code performs a communication operation like `fetch` or `wait`, the current task is suspended and a scheduler picks another task to run.
- A task is restarted when the event it is waiting for completes.

Dynamic scheduling

- For many problems, it is not necessary to think about tasks directly.
- However, they can be used to wait for multiple events at the same time, which provides for *dynamic scheduling*.
- In dynamic scheduling, a program decides what to compute or where to compute it based on when other jobs finish.
- This is needed for unpredictable or unbalanced workloads, where we want to assign more work to processes only when they finish their current tasks.
- As an example, consider computing the ranks of matrices of different sizes

```
M = {rand(800,800), rand(600,600), rand(800,800), rand(600,600)}

pmap(rank, M)
```

Implementation of pmap

Main idea

In the implementation below, a local task feeds work to each processor when it completes its current task.

```
function pmap(f, lst)
    np = nprocs()  # determine the number of processes available
    n = length(lst)
    results = cell(n)
    i = 1
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx] = remotecall_fetch(p, f, lst[idx])
                    end
                end
            end
        end
    end
    results
end
```

`@spawnlocal`, `@sync` and `@everywhere`

`@spawnlocal` (recently renamed `@async`)

- `@spawnlocal` is similar to `@spawn`, but only runs tasks on the local processor.
- In the `pmap` example above, we use it to create a feeder task for each processor.
- Each task picks the next index that needs to be computed, then waits for its processor to finish, then repeats until we run out of indexes.

`@sync`

- A `@sync` block is used to wait for all the local tasks to complete, at which point the whole operation is done.
- Notice that all the feeder tasks are able to share the state `i` via `next_idx()` since they all run on the same processor.
- However, no locking is required, since the threads are scheduled cooperatively and not preemptively.
- This means context switches only occur at well-defined points (during the fetch operation).

`@everywhere`

- It is often useful to execute a statement on all processors, particularly for setup tasks such as loading source files and defining common variables. This can be done with the `@everywhere` macro.

More on tasks

```
julia> #like @spawn, but runs taks on the local process

julia> x=@async println("hi")
Task

julia>

julia> a = @async 1+2
Task

julia>

julia> #get the calue

julia> fetch(a)
Task
```

More on remote calls (1/2)

```
julia> #make an uninitialized remote reference on the local machine

julia> r = RemoteRef()
RemoteRef(1,1,117)

julia>

julia> @async (fetch(r);println("hi"))
Task

julia>

julia> #store a value to a remote reference

julia> put(r,0)

0

julia> hi
```

More on remote calls (2/2)

```
julia> nprocs()
5


julia> #get the id of the current processor

julia> myid()
1

julia> #execute on all the processors

julia> @everywhere println(myid())
1
From worker 2: 2
From worker 5: 5
From worker 4: 4
From worker 3: 3

julia> r = @spawn myid()
RemoteRef(3,1,127)

julia> fetch(r)
3

julia> fetch(@spawn myid())
4

julia> fetch(@spawnat 2 myid())
2
```

Plan

Computing the maximum value of an array in parallel

```
julia> @everywhere function maxnum_serial(a,s,e)
                if s==e
                  a[s]
         else
                   mid = ifloor((s+e)/2)
                   low = maxnum_serial(a,s,mid)
                   high = maxnum_serial(a,mid+1,e)
                   low >high? low:high
                end
       end

julia> @everywhere function maxnum_parallel(a,s,e)
                if (e-s)<=10000000
                  maxnum_serial(a,s,e)
               else
                   mid = ifloor((s+e)/2)
                   low_remote = @spawn maxnum_parallel(a,s,mid)
                   high = maxnum_parallel(a,mid+1,e)
                   low = fetch(low_remote)
                   low > high? low:high
                end
       end

julia> a=rand(20000000);

julia> @time maxnum_serial(a,1,20000000)
elapsed time: 0.458792535 seconds (61556 bytes allocated)
0.999999919794377

julia> @time maxnum_parallel(a,1,20000000) ## two recursive calls
elapsed time: 0.654630977 seconds (268541944 bytes allocated)
0.999999919794377
```

As we can see, the parallel version runs slower than its serial counterpart. Indeed, the amount of work (number of comparisons) is in the same order of magnitude of data transfer (number of integers to move from one processor than another). But the latter costs much more clock-cycles.

Computing the minimum and maximum values of an array in parallel

```
julia> @everywhere function minimum_maximum_serial(a,s,e)
                if s==e
                  [a[s], a[s]]
         else
                   mid = ifloor((s+e)/2)
                   X = minimum_maximum_serial(a,s,mid)
                   Y = minimum_maximum_serial(a,mid+1,e)
                   [min(X[1],Y[1]), max(X[2],Y[2])]
                end
       end

julia> @everywhere function minimum_maximum_parallel(a,s,e)
                if (e-s)<=10000000
                  minimum_maximum_serial(a,s,e)
               else
                   mid = ifloor((s+e)/2)
                   R = @spawn minimum_maximum_parallel(a,s,mid)
                   Y = minimum_maximum_parallel(a,mid+1,e)
                   X = fetch(R)
                   [min(X[1],Y[1]), max(X[2],Y[2])]
                end
       end

julia> a=rand(20000000);

julia> @time minimum_maximum_serial(a,1,20000000)
elapsed time: 7.89881551 seconds (3840094852 bytes allocated)

julia> @time minimum_maximum_parallel(a,1,20000000)
elapsed time: 4.32320816 seconds (2188546996 bytes allocated)
```

In-place serial merge sort

```
julia> function mergesort(data, istart, iend)
                        if(istart < iend)
                                mid = (istart + iend) >>>1
                                mergesort(data, istart, mid)
                                mergesort(data, mid+1, iend)
                                   merge(data, istart, mid, iend)
                        end
                end
# methods for generic function mergesort
mergesort(data,istart,iend) at none:2

julia> function merge( data, istart, mid, iend)
                        n = iend - istart + 1
                        temp = zeros(n)
                        s = istart
                        m = mid+1
                        for tem = 1:n
                                if s <= mid && (m > iend || data[s] <= data[m])
                                        temp[tem] = data[s]
                                        s=s+1
                                else
                                        temp[tem] = data[m]
                                        m=m+1
                                end
                        end
                        data[istart:iend] = temp[1:n]
                end
# methods for generic function merge
merge(data,istart,mid,iend) at none:2

julia> n = 1000000

julia> A = [rem(rand(Int32),10) for i =1:n];

julia> @time mergesort(A, 1, n);
elapsed time: 0.6119898 seconds (447195104 bytes allocated)
```

Out-of-place serial merge sort

```
julia> function mergesort(data, istart, iend)
                        if(istart < iend)
                                mid = ifloor((istart + iend)/2)
                                a = mergesort(data, istart, mid)
                                b = mergesort(data,mid+1, iend)
                                c = merge(a, b, istart, mid, iend)
                        else
                            [data[istart]]
                        end
               end
# methods for generic function mergesort

julia> function merge(a, b, istart, mid, iend)
                        n = iend - istart + 1
                        nb = iend - mid
                        na = mid - istart + 1
                        c = zeros(n)
                        s = 1
                        m = 1
                        for tem = 1:n
                                if s <= na && (m > nb || a[s] <= b[m])
                                        c[tem] = a[s]
                                        s=s+1
                                else
                                        c[tem] = b[m]
                                        m=m+1
                                end
                        end
                        c
               end
# methods for generic function merge
julia> n = 1000000;
julia> A = [rem(rand(Int32),10) for i =1:n];
julia> @time mergesort(A, 1, n);
elapsed time: 0.60765198 seconds (348516200 bytes allocated)
```

Out-of-place parallel merge sort

```
..@everywhere function mergesort_serial(data, istart, iend)
                        if(istart < iend)
                                mid = ifloor((istart + iend)/2)
                                a = mergesort_serial(data, istart, mid)
                                b = mergesort_serial(data,mid+1, iend)
                                c = merge(a, b, istart, mid, iend)
                        else
                            [data[istart]]
                        end
                end

@everywhere function mergesort_parallel(data, istart, iend)
                        if(iend - istart <= 250000)
                        then
                                mergesort_serial(data, istart, iend)
                         else
                                mid = ifloor((istart + iend)/2)
                                a = @spawn mergesort_parallel(data, istart, mid)
                                b = mergesort_parallel(data,mid+1, iend)
                                c = merge(fetch(a), fetch(b), istart, mid, iend)
                        end
                end

julia> n = 1000000;

julia> A = [rem(rand(Int32),10) for i =1:n];

julia> @time mergesort_serial(A, 1, n);
elapsed time: 0.646847581 seconds (347445544 bytes allocated)

julia> @time mergesort_parallel(A, 1, n);
elapsed time: 0.391781819 seconds (125981184 bytes allocated)
```

Distributed Arrays (1/7)

Idea

- Large computations are often organized around large arrays of data.
- In these cases, a particularly natural way to obtain parallelism is
- to distribute arrays among several processes.
- This combines the memory resources of multiple machines, allowing use of arrays too large to fit on one machine.
- Each process operates on the part of the array it owns, providing a ready answer to the question of how a program should be divided among machines.

The `DArray` type

- Julia distributed arrays are implemented by the `DArray` type.
- A DArray has an element type and dimensions just like an Array.
- A DArray can also use arbitrary array-like types to represent the local chunks that store actual data.
- The data in a DArray is distributed by dividing the index space into some number of blocks in each dimension.

Distributed Arrays (2/7)

Constructing distributed arrays

Common kinds of arrays can be constructed with functions beginning with d:

```
dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x, 100,100,10)
```

In the last case, each element will be initialized to the specified value x. These functions automatically pick a distribution for you.

Constructing distributed arrays with more control

For more control, you can specify which processors to use, and how the data should be distributed:

```
dzeros((100,100), workers()[1:4], [1,4])
```

- The second argument specifies that the array should be created on the first four workers. When dividing data among a large number of processes, one often sees diminishing returns in performance. Placing DArrays on a subset of processes allows multiple DArray computations to happen at once, with a higher ratio of work to communication on each process.
- The third argument specifies a distribution; the nth element of this array specifies how many pieces dimension n should be divided into. In this example the first dimension will not be divided, and the second dimension will be divided into 4 pieces. Therefore each local chunk will be of size (100,25). Note that the product of the distribution array must equal the number of processors.

Distributed Arrays (3/7)

Constructing distributed arrays with even more control

The primitive DArray constructor has the following somewhat elaborate signature:

```
DArray(init, dims[, procs, dist])
```

- `init` is a function that accepts a tuple of index ranges. This function should allocate a local chunk of the distributed array and initialize it for the specified indices.
- `dims` is the overall size of the distributed array.
- `procs` optionally specifies a vector of processor IDs to use.
- `dist` is an integer vector specifying how many chunks the distributed array should be divided into in each dimension.
- The last two arguments are optional, and defaults will be used if they are omitted.

Example

As an example, here is how to turn the local array constructor fill into a distributed array constructor:

```
dfill(v, args...) = DArray(I->fill(v, map(length,I)), args...)
```

In this case the init function only needs to call fill with the dimensions of the local piece it is creating.

Distributed Arrays (4/7)

```
julia> @everywhere function par(I)
           # create our local patch
           # I  is a tuple of intervals, each interval is
           # regarded as a 1D array with integer entries
           # size(I[1], 1) gives the number of entries in I[1]
           # size(I[2], 1) gives the number of entries in I[2]
           d=(size(I[1], 1), size(I[2], 1))
           m = fill(myid(), d)
           return m
       end

julia>

julia> @everywhere h=8

julia> @everywhere w=8

julia> m = DArray(par, (h, w), [2:5])
8x8 DArray{Int64,2,Array{Int64,2}}:
 2  2  2  2  4  4  4  4
 2  2  2  2  4  4  4  4
 2  2  2  2  4  4  4  4
 2  2  2  2  4  4  4  4
 3  3  3  3  5  5  5  5
 3  3  3  3  5  5  5  5
 3  3  3  3  5  5  5  5
 3  3  3  3  5  5  5  5
```

Distributed Arrays (5/7)

```
julia> m.chunks
2x2 Array{RemoteRef,2}:
 RemoteRef(2,1,28)  RemoteRef(4,1,30)
 RemoteRef(3,1,29)  RemoteRef(5,1,31)

julia> m.indexes
2x2 Array{(Range1{Int64},Range1{Int64}),2}:
 (1:4,1:4)  (1:4,5:8)
 (5:8,1:4)  (5:8,5:8)

julia> @spawn rank(m)
RemoteRef(3,1,289)

julia> @spawn rank(m)
RemoteRef(4,1,290)

julia> @spawn rank(m)
RemoteRef(5,1,291)

julia> exception on 3: exception on 4: exception on ERROR: 5: ERROR: ERROR: no method svdvals(DAr
 in rank at linalg/generic.jl:87
 in anonymous at multi.jl:1239
 in anonymous at multi.jl:804
 in run_work_thunk at multi.jl:563
 in anonymous at task.jl:76
```

Distributed Arrays (6/7)

```
@spawnat 2 println(localpart(m)) ### VERSION 2.0
RemoteRef(2,1,292)

julia> mm = @spawnat 2 rank(localpart(m))
RemoteRef(2,1,293)

julia> fetch(mm)
From worker 2: 2 2 2 2
From worker 2: 2 2 2 2
From worker 2: 2 2 2 2
From worker 2: 2 2 2 2
From worker 2:
1

julia> ?DArray
Loading help data...
Base.DArray(init, dims[, procs, dist])

   Construct a distributed array. "init" is a function that accepts
   a tuple of index ranges. This function should allocate a local
   chunk of the distributed array and initialize it for the specified
   indices. "dims" is the overall size of the distributed array.
   "procs" optionally specifies a vector of processor IDs to use.
   "dist" is an integer vector specifying how many chunks the
   distributed array should be divided into in each dimension.

   For example, the "dfill" function that creates a distributed
   array and fills it with a value "v" is implemented as:

   "dfill(v, args...) = DArray(I->fill(v, map(length,I)), args...)"
```

Distributed Arrays (7/7)

Operations on distributed arrays

- `distribute(a::Array)` converts a local array to a distributed array.
- `localpart(a::DArray)` obtains the locally-stored portion of a DArray.
- `myindexes(a::DArray)` gives a tuple of the index ranges owned by the local process.
- `convert(Array, a::DArray)` brings all the data to the local processor.
- Indexing a DArray (square brackets) with ranges of indexes always creates a `SubArray`, not copying any data.

Distributed arrays and parallel reduction (1/4)

```
[moreno@compute-0-3 ~]$ julia -p 5
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.2.0-prerelease+3622
 _/ |\__'_|_|_|\__'_|  |  Commit c9bb96c 2013-09-04 15:34:41 UTC
|__/                   |  x86_64-redhat-linux

julia> da = @parallel [2i for i = 1:10]
10-element DArray{Int64,1,Array{Int64,1}}:
  2
  4
  6
  8
 10
 12
 14
 16
 18
 20
```

Distributed arrays and parallel reduction (2/4)

```
julia> procs(da)
4-element Array{Int64,1}:
 2
 3
 4
 5

julia> da.chunks
4-element Array{RemoteRef,1}:
 RemoteRef(2,1,1)
 RemoteRef(3,1,2)
 RemoteRef(4,1,3)
 RemoteRef(5,1,4)

julia>

julia> da.indexes
4-element Array{(Range1{Int64},),1}:
 (1:3,)
 (4:5,)
 (6:8,)
 (9:10,)

julia> da[3]
6

julia> da[3:5]
3-element SubArray{Int64,1,DArray{Int64,1,Array{Int64,1}},(Range1{Int64},)}:
  6
  8
 10
```

Distributed arrays and parallel reduction (3/4)

```
julia> fetch(@spawnat 2 da[3])
6

julia>

julia> { (@spawnat p sum(localpart(da))) for p=procs(da) }
4-element Array{Any,1}:
 RemoteRef(2,1,71)
 RemoteRef(3,1,72)
 RemoteRef(4,1,73)
 RemoteRef(5,1,74)

julia>

julia> map(fetch, { (@spawnat p sum(localpart(da))) for p=procs(da) })
4-element Array{Any,1}:
 12
 18
 42
 38

julia>

julia> sum(da)
110
```

Distributed arrays and parallel reduction (4/4)

```
julia> reduce(+, map(fetch,
                 { (@spawnat p sum(localpart(da))) for p=procs(da) }))
110

julia>

julia> preduce(f,d) = reduce(f,
                             map(fetch,
                                 { (@spawnat p f(localpart(d))) for p=procs(d) }))
# methods for generic function preduce
preduce(f,d) at none:1

julia>

julia> preduce(min, da)
2

julia>

julia> preduce(max, da)
20
```

Plan

Simulation 1/14

```
julia> @everywhere function SimulationSerial(A,N,T)
           for t=0:(T-1)
              past = rem(t,3) +1
              present = rem(t+1,3) +1
              future = rem(t+2,3) + 1
              for x=1:N
                A[future,x] = (A[present,x] + A[past,x]) / 2
              end
            end
        end

julia>
```

Comments

- Consider a simple simulationserial with a stencil of the form $A[t+2,i] = (A[t+1,i] + A[t,i])/2$
- We start a serial function realizing T time steps at N points.

Simulation 2/14

```
julia> N = 16
16

julia> T = 7
7

julia> A = ones(3,N)
3x16 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0

julia> SimulationSerial(A, N, T)
```

Comments

- We continue with a very simple input data for testing our serial code.

Simulation 3/14

```
julia> dA = dones(3,N)
3x16 DArray{Float64,2,Array{Float64,2}}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0

julia> for p=procs(dA) @spawnat p println(localpart(dA)) end

julia> for p=procs(dA) @spawnat p println(((dA.indexes)[p-1])) end

julia> for p=procs(dA) @spawnat p println(size((dA.indexes)[p-1][2],1)) end

julia>  From worker 9: 1 1
From worker 9: 1 1
From worker 9: 1 1
From worker 7: 1 1
From worker 7: 1 1
               ...           ...
From worker 7: :3,11:12)
From worker 8: (21
From worker 8: :3,13:14)
```

Comments

- In preparation for a parallel implementation, we review how to manipulate distributed arrays.

Simulation 4/14

```
julia> function SimulationParallel(dA,N,T)
           P = length(procs(dA))
           Nlocal = [size((dA.indexes)[w][2],1) for w=1:P]
           refs = [@spawnat (procs(dA))[w]
                   SimulationSerial((localpart(dA)),
                                    Nlocal[w], T) for w=1:P]
           pmap(fetch,refs)
       end
# methods for generic function SimulationParallel
SimulationParallel(dA,N,T) at none:2
```

Comments

- In this code, each worker updates its local part without exchanging data with the other workers
- Remote calls get workers to start computing at essentially the same time
- The last statement of the code forces the workers to complete before returning from the function

Simulation 5/14

```
julia> N = 1000000
1000000

julia> T = 1000
1000

julia> A = ones(3,N)
3x1000000 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1

julia> @time  SimulationSerial(A, N, T)
elapsed time: 4.09132303 seconds (6896 bytes allocated)
```

Comments

- Now we consider a large example with $1,000,000$ points and $1,000$ time steps.
- The serial code runs in 4 seconds.

Simulation 6/14

```
julia> dA = dones(3,N)
3x1000000 DArray{Float64,2,Array{Float64,2}}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0

julia> @time SimulationParallel(dA,N,T)
elapsed time: 0.852344034 seconds (8973220 bytes allocated)
8-element Array{Any,1}:
 nothing
 nothing
 nothing
 nothing
 nothing
 nothing
 nothing
 nothing
```

Comments

- Our first parallel function runs 5 times faster on 8 cores.

Simulation 7/14

```
julia> function SimulationParallelWithSynchronization(dA,N,T)
            Ps = procs(dA)
            P = length(procs(dA))
            Nlocal = [size((dA.indexes)[w][2],1) for w=1:P]
            for t=0:(T-1)
               for w=1:P
                  @spawnat Ps[w] SimulationSerial((localpart(dA)),Nlocal[w], 1)
               end
               for w=1:P
                  @spawnat Ps[w] (localpart(dA) [1,1] = dA[1,N])
               end
               ## the above inner loop consumnes a lot of resources
            end
        end
# methods for generic function SimulationParallelWithSynchronization
SimulationParallelWithSynchronization(dA,N,T) at none:2
```

Comments

- Now we consider a more challenging situation where synchronization (among workers) and data communication are needed after time step.

Simulation 8/14

```
julia> N = 1000000
1000000

julia> T = 1000
1000

julia> A = ones(3,N)
3x1000000 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0

julia> @time  SimulationSerial(A, N, T)
elapsed time: 4.064461911 seconds (48 bytes allocated)

julia> dA = dones(3,N)
3x1000000 DArray{Float64,2,Array{Float64,2}}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0

julia> @time SimulationParallelWithSynchronization(dA,N,T)
elapsed time: 9.485898884 seconds (1616870028 bytes allocated)
```

Comments

- This results in a severe slow-down: the new parallel code is twice slower than the serial code.

Simulation 9/14

```
julia> function SimulationParallelWithLessSynchronization(dA,N,T,s)
            Ps = procs(dA)
            P = length(procs(dA))
            Nlocal = [size((dA.indexes)[w][2],1) for w=1:P]
            for t=0:(div(T-1,s))
               for w=1:P
                 @spawnat Ps[w] SimulationSerial((localpart(dA)),Nlocal[w], s)
               end
               for w=1:P
                  @spawnat Ps[w] (localpart(dA) [1,1] = dA[1,N])
               end
            end
        end
# methods for generic function SimulationParallelWithLessSynchronization
SimulationParallelWithLessSynchronization(dA,N,T,s) at none:2
```

Comments

- Assume now that synchronization (among workers) and data communication are needed after s time step, where s is an extra argument of the function.

Simulation 10/14

```
julia> dA = dones(3,N)
3x1000000 DArray{Float64,2,Array{Float64,2}}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0

julia> @time SimulationParallelWithLessSynchronization(dA,N,T,10)
elapsed time: 1.179147254 seconds (164115192 bytes allocated)
```

Comments

- This new paralle code runs 4 times faster (than the serial code) on 8 cores.

Simulation 11/14

```
julia> N = 1000000
1000000

julia> T = 10000
10000

julia> A = ones(3,N)
3x1000000 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1

julia> @time  SimulationSerial(A, N, T)
elapsed time: 40.770635487 seconds (48 bytes allocated)
```

Comments

- From now on, `T` is multiplied by 10.
- Which multiplies the serial time by 10.

Simulation 12/14

```
julia> dA = dones(3,N)
3x1000000 DArray{Float64,2,Array{Float64,2}}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0

julia> @time SimulationParallel(dA,N,T)
elapsed time: 7.98175345 seconds (893176 bytes allocated)
8-element Array{Any,1}:
 nothing
 nothing
 nothing
 nothing
 nothing
 nothing
 nothing
 nothing
```

Comments

- The parallel time without communication is also multiplied by 10.

Simulation 14/14

```
julia> @time SimulationParallelWithSynchronization(dA,N,T)
elapsed time: 94.399175214 seconds (16163168632 bytes allocated)

julia> dA
3x1000000 DArray{Float64,2,Array{Float64,2}}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0    1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0

julia>

julia> @time SimulationParallelWithLessSynchronization(dA,N,T,10)
elapsed time: 9.869345881 seconds (1635620080 bytes allocated)
```

Comments

- The parallel time with lots of communication and synchronization is also multiplied by 10.
- The parallel time with few communication and synchronization is only multiplied by 8.