M/s

Managing distributed workloads

Language Reference Manual

Miranda Li (mjl2206)
Benjamin Hanser (bwh2124)
Mengdi Lin (ml3567)

Table of Contents

1. Introduction

M/s is a language for implementing a distributed system (master-slaves relationship). It allows a master server distribute work across slave nodes. The user defines the master function (akin to a main function), as well as jobs that can be run either locally or on slaves.

The user will define jobs that slave nodes will run, and define how to reassemble the output of these jobs back to the user. The jobs will be issued to each slave node in a round robin way. The language hides the tedious socket handling, threading, and packet serialization from the

user. The language supports primitives like int, double, string, and vector; control flow; loops in c-like syntax.

2. Lexical elements

2.1 Comments

Characters // start a single-line comment. Characters /* start a multi-line comment, and */ end a multi-line comment.

2.2 Delimiters

Semicolon delimits a statement; space, tab, and newline delimit a token.

2.3 Identifiers

An identifier is a case-sensitive ASCII string of letters, underscores, and digits where the first character must be a letter. M/s interprets an identifier based on its type. All variables of M/s are statically typed.

2.4 Data types

Primitive Types

M/s has four primitive types: int, double, boolean, and string. An integer is a two's complement signed 32-bit integer number. A double is a two's complement signed 64-bit floating point number. A boolean is a one-byte value that can either be "true" or "false". There is no implicit casting of primitive types, i.e. a one-character string cannot be casted into an integer and vice versa. We also consider string to be a primitive type; see the section on string, *infra*. All primitive types have initial values upon declaration. Int and double have an initial value of 0. A boolean is always initialized to false.

Vectors

A vector is a contiguous sequence of elements of the same type. Internally, a vector has a pointer to the beginning of the sequence and another pointer to one past the last element. A vector of *n* elements shall allocate storage for 2^(ceil(log_2(*n*))) elements, providing amortized constant time append. A vector has an initial value of an empty vector by default. To declare a vector, you must declare it in the form "vector<type>" where type is the type of the elements in the vector. Vectors of the same type can be concatenated with the "+" operator. It supports python like slicing to access a sub vector with [start: end] to retrieve the range [start, end) portion of the vector. However, we do not support assigning ranges in vectors.

Use:

```
vector<int> a = [1,2,3,4,5];
vector<int> b = [2];
print(a[0:2]); // prints "[1,2]"
```

```
print(a+b); // prints “[1,2,3,4,5,2]”
```

Strings

A string is a sequence of characters. It is *not* a vector of characters because we do not have a character type. We consider string to be a primitive type. Initial value of a string is an empty string.

Struct (C-like Struct)

A struct contains a list of variables under one name in a block of memory. The name is used to access different variables within the name via the “->” operator. M/s supports nested structs.

Jobs

A job (data type) is an entity that represents a job (thread of execution). Internally, it is implemented as a struct that contains a unique job ID and a slave ID, which represents the machine. It also contains several boolean states such as “failed,” “done,” “running,” “cancelled.” See more about these states under Jobs section. Further discussions on failures are under the topic “Error Handling”, *infra* .

Type	**Description**
int	a 32-bit integer
double	
string	Standard string
boolean	true or false
vector	As in C++
struct	As in C struct. Variables within the struct can be accessed via “->” operator.
job	● Represents a job, and also is used as the keyword to define a new job ● To assign a job: job c = remote f() ● To retrieve output: get job ● Potential fields for each job: status, return value, slave id, proc id, unique id ● Is pronounced /dʒoʊb/

3. Expressions and operators

3.1 Expressions

An expression contains at least one operand, and may contain unary or binary operators specifying operations on the operand.
All unary operators have format <*operator*> <*expression*>
All binary operators have format <*expression*> <*operator*> <*expression*>

3.2 Arithmetic operators

Binary operators: + - * / %
These operators can only be applied to ints and doubles.

3.3 Logical operators

Unary operators

- !
    - *! expr1 is true if expr1 is false, false otherwise*
    - *Only works for booleans*

Binary operators

- ==
    - *expr1 == expr2 is true if the values of the two expressions are equal, false otherwise*
    - *Works for ints, doubles, strings, booleans, strings, vectors, and jobs*
- !=
    - *expr1 != expr2 is true if the values of the two expressions are not equal, false otherwise*
    - *Works for ints, doubles, strings, booleans, strings, vectors, and jobs*
- <
    - *expr1 < expr2 is true if the value of expr1 is less than the value of expr2, false otherwise*
    - *Only works for ints and doubles*
- >
    - *expr1 > expr2 is true if the value of expr1 is greater than the value of expr2, false otherwise*
    - *Only works for ints and doubles*
- <=
    - *expr1 < expr2 is true if the value of expr1 is less than or equal to the value of expr2, false otherwise*
    - *Only works for ints and doubles*
- >=

    - *expr1 < expr2 is true if the value of expr1 is greater than or equal to the value of expr2, false otherwise*
    - *Only works for ints and doubles*
- and
    - *expr1 and expr2 is true if expr1 is true and expr2 is true, false otherwise*
    - *Only works for booleans*
- or
    - *expr1 and expr2 is true if expr1 is true or expr2 is true, false otherwise*
    - *Only works for booleans*

3.4 Assignment, =

Assigns a value to a variable.
There is different behavior for different *left hand side types*:

- Primitives, string, struct, vectors *when RHS is not* `get`*-ting from a job*
    - Performs a deep copy
    - *Use*
        - `int a = 3;`
- Primitives, string, struct, vectors *when RHS is* `get`*-ting from a job*
    - Blocks until job is finished
    - Performs a shallow copy (because the result from the job will not be used until we *get* it)
    - *Use*
        - `int a = 3;`
        - `int b = 6;`
        - `int x = get remote gcd(a,b); // blocks until gcd finishes`
- Job
    - *Does not block;* it is a lazy assignment
    - Will only block when get <job_name> is called
    - *Use*
        - `int a = 3;`
        - `int b = 6;`
        - `job c = remote gcd(a,b); // does not block`
        - `int x = get c; //blocks`
        - `print("x is ready when this statement is run");`

3.5 Vector operators, [ ]

- [x]
    - Random access operator for an element of the vector.
- [x:y]
    - Random access operator for a range of elements in the vector.

3.6 Struct operator, ->

-> is used to access variables within a struct.

- e.g. a->var

3.7 Job status operator, ->

Job is internally implemented as a struct. To access boolean statuses in a job, use the struct operator "->" See section on Job states under Jobs, *infra.*

- e.g. j->ready

4. Keywords

4.1 Control flow

while
Continues looping on a given set of statements while the given expression evaluates to true

```
while (<expression>) {
	//statements;
}
```

if
Runs a given set of statements if the given expression evaluates to true. Otherwise skips the statements

```
if (<expression>) {
	//statements;
}
```

else
After an if statement, jump to statements in else block if provided

```
if (<expression>) {
	//statements;
}
else {
	//other statements;
}
```

If, else, and while will include the following line automatically without need for brackets. Because of this, we get `else if` for free.

4.2 Printing

print is a function that takes as a primitive type as an argument and prints to the command line.

Use:

```
int a = 2
string s = “hello”
print(a);
print(s);
print(“hi there”);
// Prints 2hellohithere to console
```

4.3 Function and job definition

func

A function is a block of code which may be called as a function, but not as a job (either locally or to a slave node). They can be called from master, or called within jobs.

Use:

```
func <return_type> <function_name> ( <input type 1> <input 1>, <input type 2> <input 2>   ) {
        <function_definition>;
}
```

job

A job is a function that can be sent to slave nodes to run with remote, run on a separate thread in master with local, or directly run by master.

Use:

```
job <return_type> <job_name> ( <input type 1> <input 1>, <input type 2> <input 2>   ) {
        <job_definition>;
}
```

return

The return keyword indicates the value to return from a job or a function.

void

The return type for jobs and functions that do not return anything.

4.4 Jobs

Job states

failed

Initial state: false
failed state is set to true if and only if the job encountered network errors, runtime exceptions, or general errors. Note that cancelling a job does not set this state to true.

done

Initial state: false
done state is set to true if and only if the job has successfully finished executing and is ready to return its result.

running

Initial state: true.
running state is set to true if and only if the job is still running. Internally, running is computed by: running = (!failed ^ !done ^ !cancelled ^ exist)

cancelled

Initial state: false.
cancelled state is set to true if and only if the program uses “cancel” operator on a currently running job.

exist

Initial state: true.
exist state is set to false if and only if the program has already used “get” operator on a done job.

restartable

Initial state: false.
restartable state is set to true if and only if a job was cancelled or experienced a network failure.

Job operators

get

Get the return value from a job. Will block if job is not done.
Use:

```
job a = remote f();
int i = get a; //blocks until a is done
```

cancel

Cancels a running job. Any lazy assignments that were made by that job are reverted.
Canceling a failed, done, or non-existent job will generate an error.

Use:

```
int a = 3;
int b = 6;
job c = remote gcd(a,b); // c refers to the job running gcd
cancel c;
```

```
int y = get c; // generates an error
```

restart

Keyword specifically used to restart a restartable job. A job is restartable only if it has been canceled, or experienced a network failure.

Use

```
job a = remote f();
restart a;
```

4.5 master

Contains “server side code” written by the user that issues jobs to slave nodes. The master section of the code, indicated by the “master” keyword, (see program structure and samples section) is like the main function. The internal implementation of master functionalities (such as socket handling and asynchronous data assignment) will be done in C++ and linked against the compiled user codes.

4.6 Network-related keywords

remote

Denotes a job should be executed on a slave node. This keyword can only be used within master block.

Use:

```
// gcd with inputs a and b are run on a slave node, and assigned to variable x when ready
// Does not block
int a = 3;
int b = 6;
int x << remote gcd(a,b);
```

local

Denotes a job should be executed on the same node in a separate thread. Can be called from both the master and slave nodes.

Use:

```
// gcd with inputs a and b are run locally on master in a separate thread, and assigned to
variable y
//Blocks because we use the regular assignment, instead of the lazy assignment operator
int a = 3;
int b = 6;
int y = local gcd(a,b);
```

on

Run a job on a specific slave

```
job i = remote gcd(a,b)
job j = remote gcd(c,d) on i.slave;
```

5. Operator and Keyword precedence

In order of precedence:

1. Function calls ()
2. Parentheses '()'
3. -> [ ]
4. !
5. * / %
6. + -
7. < <= > >=
8. == !=
9. and
10. or
11. Job initialization (remote, local)
12. =
13. get cancel restart
14. Type declaration
15. control flow
16. return

6. Error handling

An exception occurs if a job (remote and local) fails and the program tries to access the variable associated with the output of the job.

```
job a = remote gcd(3,5); //remote job runs and then fails because of network error
int x = get a; //throws an exception and program crashes because user is trying to access a
and it has failed
```

To fix the above code such that the program doesn't crash, the user should check whether a is ready and if not, whether job of variable a has failed as follows:

```
job a = remote gcd(3,5); //remote job fails because of network error
int x;
if (!a->ready) { //if a is not ready
   if (a->failed) {//if job of a has failed
        //recovery codes
        x = -1;
   }
}
```

Basic information about the job's status is maintained in the job struct. There are eight possible statuses that the user can access, which are not mutually exclusive: exists, running, done, cancelled, failed (general), failed (network), failed (exception), restartable. More detailed failure

information is maintained in the job data type, and the user must create an instance of the job data type to access this information. See section on job status keywords, *supra.*

The other exceptions that may occur are divide by zero and vector access out of bounds.

7. Job Scheduling

Every time a slave joins the network, it is added to the list of slaves, and jobs may be assigned to it based on the scheduling policy (round robin). When a slave disconnects from the network, it is removed from the list of slaves, and the statuses of any jobs running on that slave are set to 'running -- failed'

8. Scope

Master, jobs, and funcs each have their own hierarchy of scope. There are no globally accessible variables accessible to any combination of master, jobs, or funcs. Each if statement, loop, and other block denoted by curly brackets has its own scope. Barring a naming conflict, variables declared higher in a hierarchy of scopes are accessible lower in that hierarchy.

9. Program Structure

Codes reside either in the master block ("master {...}"), jobs, or funcs. The master block is similar to a main function in that user codes in this block will be executed.

For the program to work, a machine must have a master binary and another machine must have the corresponding slave binary. To obtain the two binaries, simply compile the user codes with flags "--master" and "--slave". Both binaries must be of the same version for the program to work properly since both binaries should have the same job definitions.

10. Frequently Asked Questions

1. *When can the result of a "remote" job be obtained? (For example, your "remote" keyboard* [sic] *seems to be some sort of "fork" operation that immediately returns yet starts a process in the background. When can the results of that process be obtained?)*

   There is only one way to obtain the result of a job, whether local or remote: through the get operator.

   ```
   job a = gcd(10,12);
   int result = get a; //blocking call
   ```

   The result of the argument is obtained at the execution of the second line. This line is blocking. The program will block on this line until the result of gcd returns and is assigned to the variable "result".

2. *When does the system wait for processes to terminate and produce their results? What happens when a process attempts to access a variable whose value has yet to be assigned by a process running in parallel?*
   From the user's perspective, the system waits for a job to terminate when the get keyword is invoked on that job. The implementation may actually reap the job earlier, asynchronously.

3. *How, if at all, can processes communicate?*
   Jobs cannot communicate with each other in any way. Master may only communicate with jobs through arguments passed at job creation and a return value passed at job termination.

4. *How will you implement this?!?*
   Here's the rundown:

The compiler will be composed of two parts:

1) Master-Slave networking binary first written in C++ and compiled into LLVM manually (not using our written compiler)
2) The compiler that compiles everything that is user-defined (jobs, code within master, etc.) with ability to link to the master-slave networking binary when run

The compiler will create two files, master and slave, to be run on the respective machines.

Master will communicate with slaves through sockets. Master will send each slave a packet according to the following schema:

job to run (ordinal)	int
unique job id (to identify return value)	int
length	int
arguments (parsed according to signature)	`length` many bytes

Slave will send a packet back when it is done according to the following schema:

unique job id (to identify return value)	int
length	int
return value	`length` many bytes

Master will run user code, and when it blocks on data that is not ready yet, or if it encounters operator ->ready , it will poll the slave sockets and update user variables. (If this approach is

inadequate, then we will implement master with a separate thread that polls the sockets, but then we would need synchronization for the user variables.)

Slave will run one thread that listens to the socket for incoming jobs, parse each job request, and create a new thread for each job. Slave's binary will include all the code for all the jobs.

11. Context-Free Grammar

program -> ε | program masterdec | program jobdec | program funcdec
masterdec -> master { statements }
jobdec -> job (argsdec) { statements }
funcdec -> func (argsdec) { statements }
argsdec -> vardec | args , vardec
vardec -> type id
statements -> ε |statement | statements statement | statements vardecl

statement -> expr ; | return expr ; | { statements } | if (expr) statement | if (expr) statement else statement | while (expr) statement | cancel expr | restart expr

expr -> id | lit | expr + expr | expr - expr | expr * expr | expr / expr | expr % expr | expr == expr | expr != expr | expr < expr | expr <= expr | expr > expr | expr >= expr | id = expr | vardecl = expr | get expr | cancel expr | restart expr | remote id (args) | local id (args) | remote id (args) on expr | id (args) | expr and expr | expr or expr | ! expr | (expr) | + expr | - expr | expr [expr]

type -> int | double | bool | struct | job | string | vector <type>
args -> id | args , id
id -> (A-Z | a-z)(0-9 | A-Z | a-z | _)* | id -> id
lit -> (0-9)*(0-9 | \.)(0-9)(0-9)* | ".*"

12. Sample program (gcd)

```
master {
	int a = 100;
	int b = 40;
	job j1 = remote m3(a); // select a slave, tell it to create a thread of function m3 on
	                       // input a;
	job j2 = remote m4(b); // this remote job runs concurrently with the job above
	job j3 = remote gcd(get j1, get j2); // waits for both j1 and j2 to be ready
	job j4 = remote gcd(a,b); // doesn't run until j1 and j2 are ready because the line
	                          // above waits
	job c = remote m3(a);
	print(get j3, " ", get j4); // prints "20 20"
	if (c->ready) { // if job c is ready
		int output = get c; // blocking assignment from job c
```

```
                print(output); // only prints "300" if job c finished before the ~ operator
                              //checked if job c was ready
        else
                cancel c; // we're too impatient to wait for job c to finish, so kill it
}

job int gcd(int i, int j) { // function gcd takes int i and int j, returns int
        if (i - j == 0)
                 return i;
        if (i > j)
                return gcd(i-j, j); // function call, not thread creation
        return gcd(i, j-i);
}

job int m4(int i) {
        return i * 4;
}

job int m3(int i) {
        return i * 3;
}
```

13. References

Ritchie, Dennis. *C Language Reference Manual*.

Stroustrup, Bjarne. *A Tour of C++*.