M/s

Managing distributed workloads

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

Language description

M/s is suited for creating programs implementing a distributed system (master-slave relationship), distributing workload across different nodes within a system. In this language, there exist a “master” that is responsible for job load distribution, and “slaves” that are nodes that receive jobs from the master and execute these jobs.

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.

Potential applications

- For a database system that has hefty computation jobs, our language can be used to quickly code up a program that allocates jobs to different nodes within a database system
- Our language can be used to implement client-server applications where communications are done via the form of jobs

Syntax

Data types

Type	**Description**
int	Standard 32-bit integer
double	
char	ASCII character
string	Standard string
boolean	true or false

vector	As in C++
null	
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 assign the output from job c to int a: int a << c ● To do both at once: a << (c = remote f()) ● Potential fields for each job: status, return value, slave id, proc id, unique id ● Is pronounced /dʒoʊb/

Operators

- +
- -
- *
- /
- =
- %
- !
- ==
- !=
- <
- >
- >=
- <=
- // line comments
- ~: the “ready” operator, a 1 bit operator in every data type that signals if its assignment is complete
- <<: lazy assignment operator, ie assign values asynchronously

Keywords

Keyword	**Description/usage**
and	Logical and
or	Logical or
not	Logical not
while	Same as C

if	Same as C
else	Same as C
else if	Same as C
void	
return	
master	Contains “server” side code (written by the user) that issues jobs to slave nodes
remote	Denotes a job should be executed on a slave node (can only be used within master block)
local	Denotes a job should be executed on the same node in a separate thread
cancel (alias: smite)	Cancel a running job

Implementation

The compiler will be composed of two parts:

1) Master-Slave networking binary first written in a high level language (likely in C++ or Python) 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 ~ , 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.

Sample program (gcd) :

```
master {
       int a = 100;
       int b = 40;
       int c << remote m3(a); // select a slave, tell it to create a thread of function m3 on
                              // input a;
       int d << remote m4(b); // this remote job runs concurrently with the job above
       int result1 << remote gcd(c,d); // waits for both c and d to be ready
       int result2 << remote gcd(a,b); // doesn't run until c and d are ready because the line
                                       // above waits
       job c = remote m3(a);
       int output << c; // lazy assignment from job c
       print(result1, " ", result2); // prints "20 20"
       if (~output) // if output (from job c) is ready
              print(output); // only prints "300" if job c finished before the previous print
                             // statement above finished
       else
              cancel c; // we're too impatient to wait for job c to finish, so kill it
}

job (int i, int j -> int) gcd { // 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 i -> int) m4 {
       return i * 4;
}

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