Distributed Processing Made Easy: Proposal for the Creation of a Runtime Environment to Assist in Running Programs Across Multiple Systems

Colin Pronovost

2014-11-22
Abstract

I will produce a runtime environment that allows for distributed processing over a network connection to be achieved with ease. I intend to accomplish this by creating a daemon that will respond to commands sent from a controller and run code that the controller requests it to run. It will also provide a library of functions that programs on different systems can easily use to communicate and coordinate.
Introduction

Distributed processing is useful for a simple reason: many hands make light work. In other words, having more computers work on a problem reduces the amount of work each individual computer has to do, thus the process as a whole takes less time.

The aim of this project is to produce a runtime environment that will make the implementation of programs to perform distributed processing as easy as possible, therefore reducing the development time of such programs. It will accomplish this by creating a daemon that will run on the remote systems (the clients), listening for and responding to commands sent from a controller. The protocol is detailed in Appendix A. The client will accept jobs from the controller that it is able to perform, and, as soon as it has accomplished one, start another, if another exists. The daemon will run the programs it receives from the controller by calling a function in the supplied program with arguments sent from the controller. Enabling the controller to execute all or part of a program on several systems simultaneously would allow for programs to be parallelized without extensive rewriting, for instance by specifying that each individual program work on part of a larger problem, then send the program’s output back to the controller for integration into the overall solution. Dynamically loading programs to the clients will also allow for the programs to be installed on them simply by issuing a command from the controller, thereby easing the process of installing and updating the program.

The structure and properties of the runtime (in particular the dynamic loading) will allow for easy experimentation and benchmarking in order to determine if the process of parallelization is yielding significant gains. Ease of experimentation will allow it to be used as a tool for further research into the advantages of distributed processing and into which problems have parallel algorithms that compute the problem’s solution faster than their sequential counterparts.
Prior Work

There are many libraries that attempt to assist in the creation of programs that take advantage of distributed processing. I will now take a brief moment to describe them and show how my proposed system is different.

Message Passing Interface (MPI)

MPI allows for interhost communication in a distributed processing environment, similar to the system of communication I have described. It even allows for programs to be run on multiple systems over a network (Message Passing Interface, 2014) (mpiexec(1) Man Page, 2014). However, the complexity of performing some operations in MPI increases the workload of production significantly (Message Passing Interface, 2014). The system I propose will have a simple, easy-to-use, and non-intrusive interface in order to ease production and experimentation.

OpenMP

OpenMP provides support for parallelizing loops and sections of code through preprocessor directives, allowing for code to be written to run in parallel with very little effort on the part of the programmer. However, since the platform is based on sharing memory across threads, it has virtually no support for any sort of network-based parallelization, unless shared memory across the network is implemented, for instance, in the kernel, so that the distribution over the network is transparent to user-space programs (OpenMP, 2014).

Apache Spark

Spark has an extensive library and claims to be easy to use (Apache Spark, 2014). However, it lacks a native interface and does not support dynamic loading, requiring the user to SSH into all cluster systems to update programs (Apache Spark, 2014) (Running Spark on EC2, 2014). This makes it not well suited to applications that require running several different programs or rapidly changing programs.

BOINC

BOINC is a framework that provides a lot of the same features that I have proposed. It allows for a form of dynamic program loading, remote function calls, and has a task-based framework (How BOINC Works, 2014). The key difference is that my project is focused on a system for doing research with the capabilities of distributed computing while BOINC is designed more for using volunteer CPU cycles to do big data calculations (How BOINC Works, 2014).
Goals

I will create a runtime environment that allows for programs to be uploaded to and executed on multiple systems and will allow for parallel programs to be written easily. The structure is based on dividing a problem into multiple "tasks" or "jobs" that can be completed in parallel on separate machines. The architecture will consist of a "controller" which will keep track of which jobs have been completed, which are in progress, and which are pending, and of multiple "clients" which will check out and complete jobs from the controller. The client software will run as a daemon and listen for connections from the controller.

The following features will be implemented:

- The controller will be able to receive the client’s stdout and stderr and control its stdin.
- Full programs, along with their required shared libraries, will be able to be uploaded and executed with command line arguments specified by the controller.
- Shared libraries will be able to be uploaded directly, and the controller will be able to call their functions with parameters and receive the return values.
- The controller and clients will be able to post tasks that clients will then be able to check out and complete.
- A library that will allow programs to easily communicate and coordinate with each other in order to facilitate the writing of parallel programs. This library is detailed in Appendix B.

As this is primarily a research project, I will also create some test programs that will be used for benchmarking to show the decrease in time that comes from using multiple systems. One of these will be a matrix multiplication algorithm, which is a perfect candidate for parallelization since the calculation of each cell in the product matrix is independent of the others.
Research and Development Plan

(0) Write Job Queue Functions  (1) Write Network API  (2) Write Dynamic Loading Functions

(3) Write Controller-Client Communication Functions

(4) Write Library Functions

(5) Write Benchmarking and Testing Programs (i.e. Matrix Multiplication)

**What Has Been Done**

So far, I have only really been able to work on (1). It turns out that writing network code is harder than I originally thought it would be. However, it is almost done, and I should soon be able to start working on the rest of the components.

**What Needs to Be Done, and How**

Most of the project is still incomplete. What follows is a list of how I plan to accomplish each part.

- (0) will be implemented with a basic FIFO queue. Later implementations can implement priorities and other features normally associated with process scheduling, but a FIFO queue should be sufficient for now.

- (1) is mostly functional, I just have to make it a little more stable and robust.

- (2) will be implemented by borrowing code from glibc’s ld.so dynamically linked ELF interpreter.

- (3) should be easy and straightforward once (1) is stable. Most of (4) will be implemented as part of (0), (2), or (3).

- (5) will be easy and straightforward, but requires that (0-4) function before it can be implemented.
Research and Development
Environment

The University of Rochester Computer Science department has given me access to 15 computers on the department LAN. These computers are running Linux kernel version 3.5 on dual-core hyper-threaded 2.00 GHz Intel Xeon processors using the x86 instruction set. I am able to access these computers through an SSH connection through the one of the department’s publicly accessible machines.
Acknowledgments

Jim Roche, Dave Costello, and the University of Rochester Computer Science department for providing the LAN, and Professor Kai Shen for his advice on implementation.
Appendix A

The Communication Protocol

The controller’s messages are the following:

<table>
<thead>
<tr>
<th>1 byte</th>
<th>2 bytes</th>
<th>3 bytes</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGE_TYPE</td>
<td>LENGTH</td>
<td>STATUS</td>
<td></td>
</tr>
</tbody>
</table>

and may contain extra data, the length of which is an unsigned 16 bit integer specified in the MESSAGE_LENGTH field. MESSAGE_TYPE is one of the following:

- 0 request status
- 1 jobs published
- 2 exec function
- 3 run program
- 4 pipe to stdin
- 5 request stdout
- 6 request stderr
- 7 load program
- 8 load library
- 9 terminate
- 255 bad type

which have the following meanings:

- request status: requests the client’s status, which is returned in STATUS_CODE.
- jobs published: informs clients that jobs have been published. Clients should respond by requesting jobs.
- exec function: requests that the client execute a function. The data contains the function’s name, arguments, and return type.
- run program: requests that the client run the specified program with the specified command line arguments.
- pipe to stdin: requests that the client use the provided data as the stdin for the next invocation of the program.
- request stdout: informs the client that the controller is interested in the stdout of the program.
- request stderr: like request stdout.

\(^1\)Subject to change over the course of the project if the needs of the project mandate it.
• load program: requests that the client load the program contained in the data. The program is passed as an ELF file.

• load library: requests that the client load the library contained in the data. Same format as load program.

• terminate: specifies that the client should clean up by removing programs or libraries that it downloaded from the controller and closing open file descriptors, then exit.

• bad type: sent when the client sends a message with a bad MESSAGE_TYPE.

The client’s messages to the controller are the following:

<table>
<thead>
<tr>
<th>MESSAGE_TYPE</th>
<th>MESSAGE_LENGTH</th>
<th>STATUS_CODE</th>
</tr>
</thead>
</table>

and may contain extra data, the length of which is an unsigned 16 bit integer specified in the MESSAGE_LENGTH field. MESSAGE_TYPE is one of the following:

0 request status
1 request job
2 function return
3 program exited
4 request stdin
5 send stdout
6 send stderr
7 program loaded
8 library loaded
9 terminated
255 bad type

which have the following meanings:

• request status: requests the controller’s status, which is returned in STATUS_CODE.

• request job: asks the controller for a job.

• function return: informs the controller that the function that it requested the client to execute has returned and passes the controller the function’s return value if non-void.

• program exited: informs the controller that the program that it requested the client to run has exited and passes the controller the program’s exit status.

• request stdin: requests that the controller give the client data to use as stdin for the program.

• send stdout: passes the program’s stdout to the controller.

• send stderr: similar to send stdout.

• program loaded: informs the controller that the program the controller requested the client to load has finished loading.

• library loaded: informs the controller that the library the controller requested the client to load has finished loading.

• terminated: informs the controller that the client is about to exit.
• bad type: sent when the controller sends a message with a bad MESSAGE_TYPE.

STATUS_CODE may have the following values:
0 IDLE
1 BUSY
2 JOBS_AVAILABLE (from the controller only)
Appendix B

The Library Functions

The following functions will be provided on the client:

- **int getControllerSockFD()** returns the socket for communication with the controller.
- **void sendMessage(struct message* msg)** sends the controller a message. `message` contains 4 members: `uint8 message_type`; `uint16 message_length`; `uint8 status_code`; `void* data`.
- **struct message* recvMessage(struct message* msg)** receives a message from the controller. If `msg` is non-null, the message is stored in that address, which is returned. Otherwise, new memory is allocated and returned. The length of the data is checked to ensure that it is equal to the `message_length` field, otherwise the message is dropped.
- **void setMessageMask(int mask)** specifies which messages the client is interested in receiving.
- **void clearMessageMask(int mask)** clears the message mask so that all messages are received.

The following functions will be provided on the server:

- **int getNumClients()** returns the number of clients.
- **struct client_data* getClientData(int client, struct client_data* cldata)** retrieves data on the specified client, such as OS, architecture, clock speed, amount of RAM, and other useful information. If `cldata` is non-null, the data is stored in that address, which is returned. Otherwise, new memory is allocated and returned.
- **void sendMessage(struct controller_message* msg)** sends a client a message. `controller_message` is the same as for the client, except that an `int client` field is added.
- **void sendAllMessage(struct message* msg)** sends all clients a message. `message` is the same as for the client.
- **struct controller_message* recvMessage(struct controller_message *msg)** receives the next message. If `msg` is non-null, the message is stored in that address, which is returned. Otherwise, new memory is allocated and returned. The length of the data is checked to ensure that it is equal to the `message_length` field, otherwise the message is dropped.

\[1\] Again, these are subject to change based on the needs of the project.
References


http://en.wikipedia.org/wiki/Message_Passing_Interface

mpiexec(1) man page. (2014). web. Retrieved from


http://spark.apache.org/docs/latest/ec2-scripts.html