ABSTRACT
Computing instruction typically lags “state of the art” hardware and software development. With the advent of dual and quad core processors, parallel processing has become common; even economy lap top computers are dual core. Students need to be instructed on how to take advantage of this processing power. A common and easily developed software tool is the invocation of remote methods communicated via a threaded master. This paper shares the experiences of a project implemented by a student that compares compiled, hybrid, and scripted languages for distributing data among workers. The paper suggests projects of this type are easily within the grasp of students in a Data Structures or more advanced classes.

KEY WORDS
Parallel Algorithms, Student Projects, Parallel Processors

1. Introduction
Computing education should now focus on distributed processing. Processing speeds continue to improve, but computing demands exceed our most capable computers. Students need to be taught how to take advantage of the parallel hardware component in standard computers. Rather than initially teaching students to develop algorithms in a sequential manner, students should be encouraged to create parallel algorithms which can consequently take advantage of the additional computing power presented by symmetric multiprocessors, dual and quad core machines. Classes that have introduced method invocation can immediately discuss distributed methods and distributed algorithms such as sorting or searching. The mechanism of communicating between two machines over the network or within one machine has developed so that it can be incorporated in the undergraduate curriculum. Using the C/C++ programming language’s remote procedure call (RPC), Java’s remote method invocation (RMI), or Ruby’s distributed method invocation (DRb), the student can compare compiled, hybrid and scripting language’s techniques and tools for parallel distributed program implementation.

This paper describes a project to compare C, Java, and Ruby in a similar manner by implementing a common sorting algorithm, the bubble sort, on remote machines. We timed the passing of data to a function (method) in each language environment. The student experienced compiled versus hybrid versus interpreted language efficiencies. The paper organization is as follows: section 2 describes the project in detail, section 3 describes the environment in which each implementation occurred, section 4 shows efficiency results, section 5 comments on the student experience, section 6 has conclusions and comments and section 7 references.

2. Project
The C programming language is common among all operating systems and hardware. RPC is available so that a main routine can invoke a function on either another processor within the same machine or on a remote machine connected via the network. If two cpu’s are used we would expect the problem to be solved in nearly half the time. Similarly, Java’s RMI and Ruby’s DRb provide convenient mechanisms for invoking methods that reside on remote machines. The purpose of the project is to compare the relative efficiency of C/C++’s RPC, Java’s RMI, and Ruby’s DRb for distributing data, sorting, and returning the data to the master. The student experienced implementation and efficiency characteristics of compiled, hybrid, and interpreted languages.

Each program is implemented in the following manner:

Client code:
Step 0 – Between 1000 and 128000 integers were generated one at a time. That is, a test of 1000, 2000, 4000, 8000, 16000, 32000, 64000, and 128000 total integers were run in individual tests.
Step 1 – A timer was started.
Step 2 – Communication was established between the master and 1 or 4 workers as shown in Figure 1.
Step 3 – the integers were evenly distributed between each of the workers.
Step 4 – client waits.
Step 5 – processed data received from each worker.

This paper describes a project to compare C, Java, and Ruby in a similar manner by implementing a common sorting algorithm, the bubble sort, on remote machines. We timed the passing of data to a function (method) in each language environment. The student experienced compiled versus hybrid versus interpreted language efficiencies. The paper organization is as follows: section 2 describes the project in detail, section 3 describes the environment in which each implementation occurred, section 4 shows efficiency results, section 5 comments on the student experience, section 6 has conclusions and comments and section 7 references.
Step 6 – additional processing of data by the client may be required.
Step 7 – the timer is stopped and elapsed time reported.

Server code:

Step 0 – waits.
Step 1 – receives data.
Step 2 – processes data.
Step 3 – returns data to the client.
Step 4 – waits.

3. Hardware and Software

The hardware chosen for the research consisted of a networked collection of homogenous Intel based PC’s running at 3.3 GHZ connected by gigabit fast Ethernet. Each of the machines is a dual processor with 2 gigabytes of memory.

Each machine in the cluster ran Centos Linux base version 2.6.18-92.1.13.el5d, C version 4.1.2, Ruby version 1.8.6, and Java version 1.6.0_03 to support the project. Figure 1 shows the general configuration of the machines.

Table 1 shows the results of one worker with the O(n^2) algorithm (all times in seconds).

<table>
<thead>
<tr>
<th></th>
<th>Ruby</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1k</td>
<td>0.51933</td>
<td>0.775</td>
<td>0.000403</td>
</tr>
<tr>
<td>2k</td>
<td>2.01037</td>
<td>1.005</td>
<td>0.091612</td>
</tr>
<tr>
<td>4k</td>
<td>8.36326</td>
<td>1.273</td>
<td>0.047440</td>
</tr>
<tr>
<td>8k</td>
<td>30.54568</td>
<td>1.964</td>
<td>0.195169</td>
</tr>
<tr>
<td>16k</td>
<td>115.79786</td>
<td>2.272</td>
<td>0.502247</td>
</tr>
<tr>
<td>32k</td>
<td>446.41688</td>
<td>4.167</td>
<td>1.901856</td>
</tr>
<tr>
<td>64k</td>
<td>1661.63901</td>
<td>9.512</td>
<td>6.995275</td>
</tr>
<tr>
<td>128k</td>
<td>6293.89820</td>
<td>28.532</td>
<td>27.896351</td>
</tr>
</tbody>
</table>

4. The Project Results

The testing consisted of distributing 1000 to 128000 integers between 1 and 4 workers. Sorting using a O(n^2) algorithm was tested to determine efficiency of algorithms composed of changing work requirements. The actual work being performed is unimportant since rate of growth is the primary concern along with the student experience.
Table 2 shows the results of four workers with an $O(n^2)$ algorithm (all times in seconds).

**5. Student Experience**

The student understood each algorithm as implementation and distribution of the data occurred in a similar manner.

As one may expect, Ruby is the easiest to implement, Java slightly more challenging and C/C++ the most challenging. This is consistent with language design in that Java and Ruby are newer languages than C/C++. Ruby’s execution is also the easiest with threads existing in a very natural manner.

The Java client does a lookup to find the worked with a statement like:

```java
c[0]=(Sort)Naming.lookup("//compute-0-0.local/sort");
```

and the invocation is:

```java
ret=c[0].doSort(x);
```

The Ruby code is similar, but even simpler:

```ruby
mySort=DRbObject.new(nil,"druby://compute-0-0.local:9000")
```

and the invocation is:

```ruby
crr=mySort.bin(arr).
```

Finally the C invocation, which is considerably more complex than the preceding two, takes nine arguments and a call like:

```c
Client Side:

rpc_call (char *host /*Name of server host */,
u_long prognum /* Server program number */,
u_long versnum /* Server version number */,
xdrproc_t inproc /* XDR filter to encode arg */,
char *in /* Pointer to argument */,
xdrproc_t outproc /* Filter to decode result */
char *out /* Address to store result */,
char *nettype /* For transport selection */);

Server Side:

rpc_reg(
    u_long prognum /* Server program number */,
    u_long versnum /* Server version number */,
    u_long procnum /* server procedure number */,
    char *procname /* Name of remote function */,
    xdrproc_t inproc /* Filter to encode arg */,
    xdrproc_t outproc /* Filter to decode result */,
    char *nettype /* For transport selection */);
```

<table>
<thead>
<tr>
<th></th>
<th>Ruby</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1k</td>
<td>0.0425109</td>
<td>0.409</td>
<td>0.000063</td>
</tr>
<tr>
<td>2k</td>
<td>0.1388990</td>
<td>0.892</td>
<td>0.000134</td>
</tr>
<tr>
<td>4k</td>
<td>0.4969550</td>
<td>0.906</td>
<td>0.000414</td>
</tr>
<tr>
<td>8k</td>
<td>1.8678629</td>
<td>0.937</td>
<td>0.091683</td>
</tr>
<tr>
<td>16k</td>
<td>6.9931728</td>
<td>0.957</td>
<td>0.004746</td>
</tr>
<tr>
<td>32k</td>
<td>25.581942</td>
<td>1.119</td>
<td>0.195170</td>
</tr>
<tr>
<td>64k</td>
<td>95.973659</td>
<td>1.719</td>
<td>0.502409</td>
</tr>
<tr>
<td>128k</td>
<td>378.958943</td>
<td>3.523</td>
<td>1.991898</td>
</tr>
</tbody>
</table>
Fortunately rpcgen() was created to help with the obscurity of these functions. Rpcgen will create the proper header file, and both the client and server stubs needed to handle the calls between the machines. In its simplest design, the student can use this tool to create and pass strings of characters. It is not as convenient to pass arrays of numbers. The array must be wrapped in a structure and the structure then passed. While the speed of C is unmatched by the other two the preferred method of invocation would be done in either Java or Ruby. However, the student surmised that a good reason for invoking a process on multiple machines is to speed up the processing of data, if this is the case C should be used as you will get C inherent speed.

With a little initiation, students can implement parallel code; this should be done in a natural way in all courses above and including Data Structures. This student was able to learn the necessary procedures independently through a Directed Independent Course. Any student would be able to implement simple algorithms in parallel with minor guidance.

Student response to the experiment was overwhelmingly positive. Some of our students indicated application of the concepts immediately in other areas beyond those presented in this paper. Based on this study, we believe students would find projects that utilize parallel algorithms interesting and rewarding.

6. Conclusion

This study supports the concept of teaching parallel algorithm development to undergraduate students. Our student was able to master the concepts of parallel communication in three programming language environments. Performance characteristics were demonstrated between scripting (interpreted), hybrid, and compiled languages. As shown, Ruby was slow as a result of simply processing issues. Java initially depended upon establishing communication; only after data sizes of about 32k did the algorithm become a factor. C performed well in all cases. Clearly, these kinds of projects can and should be incorporated as standard projects in Data Structures and Algorithm Analysis courses. Any algorithm where large quantities of data are present are candidates for using parallel processing. Such algorithms as searching, sorting, averaging, finding maximum or minimum are candidates for distributed processing. Rather than initially train students to develop sequential solutions, educators can focus on relatively simple parallel solutions. The paper demonstrates such process is possible and well received by the students.

References


