Interfacing JavaScript and MPI

Dr. Kalyan S. Perumalla & Charles “CJ” Johnson

Introduction

Terminology

- C: the C Language; A structured, procedural programming language that has been widely used both for operating systems and applications and that has had a wide following in the academic community. [1]
- Fortran: FORTRAN; Formula Translation; a third-generation programming language that was designed for use by engineers, mathematicians, and other users and creators of scientific algorithms. It has a very succinct and spartan syntax. Today, the C language has largely displaced it. [2]
- C++: A general-purpose, object oriented programming language developed by Bjarne Stroustrup, and is an extension of the C language. It is considered to be an intermediate level language, as it encapsulates both high and low level language features. Initially, the language was called 'C with classes’ as it had all properties of C language with an additional concept of 'classes’. However, it was renamed to C++ in 1983. [3]
- JavaScript: Javascript; JS; A scripting language primarily used on the Web. It is used to enhance HTML pages and is commonly found embedded in HTML code. JavaScript is an interpreted language. Thus, it doesn't need to be compiled. JavaScript renders web pages in an interactive and dynamic fashion. This allowing the pages to react to events, exhibit special effects, accept variable text, validate data, create cookies, detect a user’s browser, etc. [4]
- MPI: the Message Passing Interface; A de facto standard for communication among the nodes running a parallel program on a distributed memory system. MPI is a library of routines that can be called from Fortran and C programs. MPI's advantage over older message passing libraries is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimized for the hardware it runs on). [5]
- Open MPI: OMPI; Open Message Passing Interface; Open MPI is an open source, freely available implementation of both the MPI-1 and MPI-2 documents. The Open MPI software achieves high performance; the Open MPI project is quite receptive to community input. [6]

- MPI-1 and MPI-2: The MPI standard is comprised of 2 documents: MPI-1 (published in 1994) and MPI-2 (published in 1996). MPI-2 is, for the most part, additions and extensions to the original MPI-1 specification. [7]
- Node.js: Nodejs; NodeJS; Node; A server-side platform wrapped around the JavaScript language for building scalable, event-driven applications. This is confusing for even experienced programmers because the traditional JavaScript environment has always been client-side - in a user's browser or in an application that is talking to a server. JavaScript has not been considered when it comes to the server responding to client requests, but that is exactly what Node.js provides. Node.js is not written in JavaScript (it is written in C++) but it uses the JavaScript language as an interpretive language for server-side request/response processing. In other words, Node.js runs stand-alone JavaScript programs. [8]

Motivations

One of the main issues with MPI as it was defined in the MPI-1 and MPI-2 standard is its use of traditional function calls. A function (ex: `MPI_Init(&argc, &argv);`) is invoked and it is a requirement of the implementation that before the code may progress, the function must be returned. Often times this seems rather out of place due to the way the arguments are structured in the MPI library. MPI doesn't directly return values like many programmers expect. (ex: `int rank = MPI_Comm_rank(MPI_COMM_WORLD);/* doesn't work */`) Instead the specification is such that you must first allocate some memory for the data you want by defining an empty variable (ex: `int rank;`) and then passing the memory address of that variable as an argument of the function. (ex: `int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);/*rank is now set properly*/`) This makes sense, since it allows for less invocation layers on the stack, but it carries with it the baggage of directly setting variables. Before `MPI_Comm_rank()` ever returns, (returning is required in order for the code execution to progress) it must wait until the rank variable can be set. This process by which the program is forced to wait for a return statement is called "blocking" and MPI uses these for all parts of the library. "Blocking function calls" are great in that they ensure information (on an individual process) moves linearly. When a function is called, the state of the program is known and the same state is retained for when the function returns. This is a convenience and it helps for the logic of the program because it allows for more linear thinking (at the process level – it should be noted that multiple processes run independently so the same rules do not apply at that level). The reason this is a problem, though, is that it creates these gaps in "real" time (the time on your watch) where the process is perfectly capable of computing more data or performing some other action, but the system isn't allowing it to. The best example is with `MPI_Recv()`. When a process knows it will want some data from another process, it calls the "receive" function. Often times, the function is called but the data it's looking for isn't there yet. Until the accompanying `MPI_Send()` is called by another process, the data simply doesn't exist yet and the code execution is stuck. This is somewhat solved, though, as there are two "non-blocking" versions of those same functions. `MPI_Irecv()` and `MPI_Isend()` immediately return after being called. The thing is, these still have a lot of overhead to deal with when implementing them in a program. The receiving process needs to use the `MPI_Wait()` and `MPI_Probe()` functions to try and resync the data transmission process as well as learn about the messages being received in order to make a decision about whether the data is desired or not. This requires constantly looping and checking in addition to the program that's already running. This problem also is magnified by the fact that every single process must go through the same steps on their own (it can't be centralized). This is where JavaScript comes in. JavaScript is a fully interpreted language. It cannot be compiled to binary and run stand-alone like C/C++ programs can. Instead, it requires an environment in which to run. Traditionally, this is the browser; however, outside of that there is an interpreter called NodeJS. Node takes in JavaScript code and runs it in real time on any computer that can handle it. (It's compatible with GNU+Linux, Solaris, OS X, and Windows.) Part of the JavaScript specification requires an event system. Events aren't handled by JavaScript itself, but instead by the interpreter. There many predefined events in the browser such as onclick, onkeydown, and onload. Node follows this same system; however, many of these UI type events don't exist due to it being a command-line only application. Interestingly, though, events can be defined by the developer in code. A common one is an HTTP request listener. There are several libraries for running an HTTP server in Node.js. All of them create an event that fires whenever a request is received and then subsequently fire a callback function at that time. This event firing is all handled by the interpreter and is native to JavaScript. There is no need to write a loop checking for data. There's no need to constantly sniff messages and see if it contains data you're looking for. It all just happens at whatever time the data becomes available. Another feature native to JS interpreters is multithreading. JavaScript itself has no notion of multithreading in the traditional sense; however, when an event fires, instead of interrupting the execution of the currently active script, it simply spawns a new thread to run the newly started code in. There is no hard limit to how many open threads an application can have and since the memory is all managed with a garbage collection system, the programmer is allowed much flexibility in how they structure event handlers. So much so that in most well written applications, there is very little "main" code. Almost everything is tied to firing of events. This presents a massive speed improvement as well as an ease of software development simply not found in traditional languages such as C. Going back to the `MPI_Send()` and `MPI_Recv()` example, in a multithreaded, event-driven language such as JavaScript, the program flow would be very different. At some point during the initialization of the program, on the receiving process, an event would be created and assigned to a callback function. (pseudo code ex: `addEventListener(MPI_Recv(/*args*/), function(e){/*code to run*/})`) Then, the program would continue executing as normal, but at any point in the future if a message is received, the interpreter would spawn a new thread and execute the callback function. This all sounds great; however, it does lead to an intereating question. "If NodeJS is spawning new threads to run callback functions, how does communication between threads work?" The answer is quite simple in this case. "It doesn't." or better wording "It doesn't have to." Even though new threads are being spawned, this actually does not affect communication between threads. There is simply no need for it because all threads on an individual process are run using the same memory and the scope follow the flow of the document that they're written in. If an event handler is written inside a certain scope and has access to a certain set of variables, if those variables have maintained state up to the point of the event firing, then it still has access to them. It can be hard to keep track of such data so it's better to implement event handlers on the global scope, but that's simply for best practices. There's nothing about the language that would require it. If I have a variable named "test" and I have two event handlers that can see it in their scope, then both of them have full access to it just as if they were in the same thread. It is this combination of native language features that makes JavaScript so attractive. The power, the flexibility, and the ease of writing for the development process. It does add a layer of abstraction over the native code, but the potential gains far outweigh that. That is the real reason, the motivation, behind interfacing JavaScript and MPI for use in HPCs.

Technical Approach

In order to interpret the Javsacript code, Node.js was chosen. Node.js is a modified version of the Javascript engine from the Chrome web browser and it runs like any other C/C++ program in the console. You feed it the name of a Javascript file as an argument and it simply begins with the first line for excecution instead of requiring a main function. It also includes a module loader called require.js which is used to pull in more Javascript files for excecuting on the fly. There is no compile or linking process, so all files must be present and in their expected file paths in order for the it execute. Node does offer a nice feature, however. It has it's own package manager called "Node Package Manager" or NPM that allows you to download dependencies from their repo. With this, it creates a folder called node_modules that acts like the /bin folder so Node always knows to check it for any files that require.js is requesting.

Open MPI (short of "Open Message Passing Interface") is a library written in both C and Fortran that is designed for moving data between processes of a program via "messages". These messages are implemented in the form of function calls. The sending process calls a send function and the receiving process calls a receive function. The receiving process must know/expect messages to be sent to it at any time.

"Foreign Function Interface" or FFI is a Nodejs module that acts as the "glue" layer between Javascript and native C. It is provided the name of a C shared object file and it dlopen's it at runtime. There is not a way to statically link C libraries; however, it does allow for full compatibility between the two languages. C functions can be called from Javascript no problem. The returnable data is also passed back into Javascript with ease. FFI even has additional smaller modules that allow for pointers to be used in Javascript so as to allow any C code to work out of the box.

Challenges Faced

After much testing, it was determined that a variable in Open MPI was not compatible with Node.js's environment. Some value called `*best_component` was being cleared during Node.js's initialization preventing Open MPI from properly initializing. This was solved temporarily by editing Node.js's source code and adding a line that called `MPI_Init()` from inside C instead of from Javascript.

Status

The project is still created in such a way that it requires the source of Node.js and Open MPI to be edited. Ideally it would run without needing to change anything about those two codebases. More testing needs to be done, but it is believed to not be as fast as native code due to all of the layers of abstraction between Javascript and Open MPI's C code.

Future Work

In the future, a production-ready version would vastly improve the practicality and usefulness of this project.

Install Node.js

```
INSTALL OPEN MPI WITH NODE.JS COMPATIBILITY

Assumptions:
 - Debian-based Linux OS (min ver: 7.8.0)
 - sudo command pre-installed
 - Internet access to download files
 - At least 1gb of free space

Installation Steps:
 - Install build-essential
   $ cd /home/user
   $ sudo apt-get install build-essential
 - Install Open MPI
   $ cd /home/user
   $ wget http://www.open-mpi.org/software/ompi/v1.8/downloads/openmpi-
1.8.5.tar.gz
   $ tar -xvf openmpi-1.8.5.tar.gz
   $ cd openmpi-1.8.5
   $ ./configure
   $ make
   $ sudo make install
- Install Node.js and Node Package Manager
   $ cd /home/user
   // Note: the following command must not be executed prior to
configuring MPI
   // For reasons I can’t explain, it causes MPI to fail compiling.
But, once
   // ./configure is run on MPI, you can feel free to in stall these
other
   // packages since make and make install run just fine.
   $ sudo apt-get install git-core openssl libssl-dev pkg-config
   $ wget http://nodejs.org/dist/v0.12.4/node-v0.12.4.tar.gz
   $ tar -xzf node-v0.12.4.tar.gz
   $ cd node-v0.12.4
   $ (export CC=/usr/local/bin/mpicc; ./configure)
   $ vi src/node_main.cc
   // Insert “MPI_Init(&argc, &argv);“ into both main() and wmain()
   $ make
   $ sudo make install
- Install Node’s dependancies
   $ cd /home/user
   $ sudo npm install node-gyp ffi ref ref-struct ref-array

From here, you can execute ”mpirun node <filename>.js” from the user’s
home folder
and it should work just fine.
```

Install Our MPI.js

```
OPENMPI WITH NODE.JS
- Install build-essential
   $ cd /home/user
   $ sudo apt-get install build-essential git-core openssl libssl-dev
pkg-config

 - Install Open MPI as usual

 - Install Node.js and npm from nodejs.org
   $ cd ~/Downloads/node-...
   $ mkdir $HOME/nodejs
   $ ./configure --prefix=$HOME/nodejs
   $ make
   $ make install (no sudo needed because it installs to ~/nodejs)

 - export NODE_PATH=$HOME/nodejs

 - Specially compile node/src/node_main.cc to invoke MPI_Init from its
main()
   $ Edit src/node_main.cc to insert “MPI_Init(&argc, &argv);“ into
main() and wmain()
   $ Compile src/node_main.cc with mpicc
   $ Link manually, replacing gcc with mpicc for the link command
   $ make install
 - Generate mpi.js
   $ sudo apt-get install clang-dev
   $ ln -s /usr/lib/x86_64-linux-gnu/libclang-3.4.so ./libclang.so
   $ ~/nodejs/node_modules/.bin/ffi-generate -f
/usr/local/include/mpi.h -l /usr/local/lib/libmpi.so -L . > mpi.js
   $ Edit mpi.js to change "exports./usr/local/lib/libmpi.so" to
"exports.mpi"
   $ mv mpi.js $NODE_PATH/node_modules

 - Install Node's dependancies
   $ cd $NODE_PATH
   $ sudo npm install node-gyp ffi ref ref-struct ref-array

 - Run mpitest ("var mpi = require('mpi');
mpi.mpi.MPI_Init(null(null,null); mpi.mpi.MPI_Finalize();"
   $ Sequential: $NODE_PATH/bin/node mpitest.js
   $ Parallel: mpirun $NODE_PATH/bin/node mpitest.js
```