Abstract—Distributed computing is the use of distributed systems to solve computational problems. By dividing a problem into tasks which can be completed concurrently by networked computers, distributed systems make computationally challenging problems easier to solve. When writing programs to take advantage of distributed computing resources it can be hard to navigate low level socket calls and message passing. To avoid this a framework and library, written in C# have been created with the aim of making it easier to write and understand programs that take advantage of a networked system of computers using the .NET Core platform.

I. INTRODUCTION

As computing systems get larger the scale and complexity of problems we wish to solve grows accordingly. Often it is easier to find raw computing power than it is to understand how to split up a computationally challenging problem into more manageable tasks. Large scale enterprise systems exist that can manage warehouses full of hardware efficiently and effectively. These systems handle essential distributed computing challenges such as fault tolerance, redundancy and scalability.

Where they are less effective is in teaching and explaining how to solve common types of problems that are well suited for a distributed or parallel environment. To differentiate terms, parallel computing generally deals with processing units that have some type of shared memory. Processors are then able to communicate by exchanging information utilizing these shared memory resources. Distributed computing on the other hand deals with processors that have their own memory and must communicate via message passing across a network. This distinguishes a distributed system from a single machine with many cores. While the two ideas of parallel and distributed computing are not mutually exclusive for the most part this project will focus on the distributed side of things.

To teach distributed computing techniques in a way that students can relate to a system needs to be in place that takes away some of the challenging aspects of building the underlying system while being flexible enough to allow them to write code that is not tightly coupled with any architecture dependent aspects such as the number of nodes. A student should not need to worry about communicating with an underlying system itself but should only need to following an API specification to run a program.

To accomplish the task of creating a system that helps enforce common principles of distributed computing in an academic setting a library written in C# .NET has been created. With the recent addition of .NET Core to the .NET platform, cross-platform code can now be written [1]. While this is not new to languages such Java, frameworks have previously been written to accomplish a similar task; see PJ2 [2]. What makes .NET Core different is that it supports multiple languages and paradigms. C#, an object oriented event driven language, was chosen for the implementation of the framework and library, however any language within the .NET Core platform can be used to write code and it will compile and run. Currently those languages are limited to C# and F# with visual basic coming soon but even the addition of F#, a more functional language, adds flexibility to the developer and is more interesting to work with from an academic point of view.

The idea behind separating the framework from the library is to abstract away the low level details that go into building a distributed system such as socket manipulation, message protocols and communication between nodes. While important concepts in their own right, they distract developers from concepts involving the structuring of programs to complete computational challenging problems using a distributed system. The framework consists of code used to setup backend nodes, communicate between hosts and transmit data such as files and other message protocols. This code is hidden and inaccessible to users of the library, unless compiled by the user themselves. The library contains classes and code to facilitate developing a distributed program. This includes classes to set up jobs with tasks and change the behavior of how the backend nodes break up and complete the work.

II. BACKGROUND

A. Running with .NET Core

.NET Core functions differently than other standard .NET framework projects and code. Setting up and running a project has different implications. A substantial difference is ability to run software on multiple operating systems. The cross platform nature of .NET Core allows it to effectively serve as a platform for writing distributed code that can take advantage of hardware being managed by a variety of operating systems. Behind the scenes .NET Core uses NuGet as a package manager to install different dependencies. Pip in python functions in a similar fashion as do other Linux package managers like yum. For example, if your project needs "System.Threading.Thread", you would specify that in the dependencies section and when run NuGet will make sure that package is available.
The program that is used to compile and run code, aptly named, is "dotnet". dotnet builds and then runs an instance of your program in the background on the target system. This tool is command-line based and in general the system being developed will deal solely with command-line applications. The .NET Core does support other models such as ASP.NET but they are outside the scope of this project. Two terms to understand when deploying .NET applications are Framework-dependent deployments (FDD) and Self-contained deployments (SCD).

Framework-dependent deployments only involve your code along with any third party libraries. They use the version of the .NET Core that is present on the target machine. This is useful if you want a small deployment that does not need to worry about the operating system that it is being deployed too. The downside of this approach is that you need the correct version of the .NET Core installed for your application to work correctly. This means if you build an application for .NET Core version 1.0 that version needs to be installed properly on the target machine.

Self-contained deployments contain everything mentioned above and also the .NET Core version your application is running with. SCD need to specify all the target operating systems that will be used to run the application. This is a good approach if the target system does not have .NET Core installed however these deployments tend to be on the larger side because of this.

For this project a framework-dependent approach was used since, working in an academic setting, we have the advantage of setting up the proper .NET Core installation all target machines. As newer versions of the .NET Core are released they can be easily installed and the library can be recompiled against them to support a wider number of features.

B. Parallel Code

While distributed systems are a useful tool for completing tasks often times we want to build upon that by also writing parallel code. The reason that the framework and library are not distributed and parallel is that C# provides a compressive, easily accessible parallel library as part of the .NET framework which can be used in the .NET Core framework as well. This library is called the Task Parallel Library. It is based on the idea of a task or unit of work. A task can be compared to a thread and may be run as such behind the scenes. This idea of splitting larger tasks or jobs up into a more manageable size occurs on the distributed side of things as well. How those tasks are broken up is dependent upon the type of problem being worked on, however there are common solutions that are used frequently when load balancing.

III. Related Work

This project is related to the work that Professor Alan Kaminsky has done in the area of Distributed Systems. His Parallel Java 2 library and book [2] contain ideas that are used in this distributed C# library specifically as it relates to job and task schedules and his pattern for writing networked systems was the starting point for the client server architecture described in the following section.

IV. Architecture

At a basic level the architecture for this system is client-server. A server is setup to manage three different types of clients. The first and largest type are the node clients. These are programs running on different machines networked with the server that handle execution of user code as well as error handling and results reporting. They do the majority of work within the system where the server simply manages connections and sends result back and forth. The second type of client is a user of the system. Users submit jobs to server who breaks up the work and distributes it to some number of clients based on the requirements of the users program.

The last type of client combines all the other communication into a series of different commands, such as 'query', which would allow someone to gain additional information about the system but not maintain a persistent connection to the server itself.

To manage the scheduling of user jobs to available resources a scheduler task is available and prioritizes jobs based on the requested nodes and the number of nodes available. The scheduler communicates directly with the server letting the server known when to send jobs to nodes. When a user job connects to the system a reference is first sent to the scheduler and then the scheduler communicates back to the server who sends a copy of the users program, which is compiled into a dynamically linked library (DLL) beforehand, to each node performing part of the distributed computation. Each node then creates a separate process for the users program and manipulates it using reflection to perform the necessary tasks before sending the server back resulting information.

V. Implementation

The entire system is built upon the .NET Core platform and written entirely in C#. The .NET Core release version

![Fig. 1: System Diagram](image-url)
used is 1.1. This version was installed on all of the Computer Science Linux machines at the Rochester Institute of Technology. DEFLib is the name of the library built for this project. The following subsections are broken down by namespace (alphabetical order) to give an overview of the components involved. From here on out any mention to “the distributed system” refers to the distributed system created by the C# framework and library described by this paper.

A. Distributed.Assembly

The components inside the assembly namespace are used to interact with a users program for the purpose of giving the appearance of running their code as written while controlling the actual process behind the scenes. The namespace contains two files; AssemblyLoader and JobExecuter. AssemblyLoader contains the Core Loader class which takes in a compiled program as a DLL and attempts to find the type specified by the generic parameter. The class declaration looks like the following:

```csharp
public class CoreLoader<T>
```

It attempts to find the class type by looping over every type specified by the DLL and creating an object that is of the parameterized type T. If the creation of the object fails then it is not of the required type and an exception is thrown. If no types matching the parameterized type are found an exception is thrown. Upon successfully initializing an object of the requested type the CoreLoader object can be used to call methods out of the assembly and also access properties. This class is essential to how the system operates. Because a users program is required to implement specific interfaces detailed in the Distributed.Jobs section, the nodes are able to call methods of the inheriting class without knowing anything about the actual implementation. This level of flexibility allows for any number of generic programs to be run through the system.

The second file within the namespace is the JobExecuter (with corresponding class of the same name). This class is responsible for the proper execution of a users program. Each node in the cluster will run this class to perform the users tasks. In order to allow the user to “print” output to standard out the JobExecuter swaps the output text writer for a custom text writer that stores user output so that it can be piped back at a later point in time. The output stream works exactly the same way and has the same feel as the standard output stream but since code could be running on any number of machines it does not serve the user to have their output displayed to that machine. The executor instantiates a CoreLoader<T> object and calls the Main method passing in any arguments the user specified. Then based on the schedule specified by the user (see schedule details in Distributed.Jobs) the JobExecuter executes the main method for each corresponding task given to the current node to complete. On success the JobExecuter sends back the users output and results to the node who communicates back to server and ultimately the user themselves.

B. Distributed.IO

Distributed.IO contains the functionality to perform file reading and writing operations across a network as well as the custom text writer used by the JobExecuter to store user output before sending it back. The file reading and writing are used to send a compiled dll from the user to the server and then to each node being used for the job. It blocks on the network stream and only that file is read or written until completion unlike standard messages which are queued up when they are received. The custom text writer relays the information in JSON format back to the clients.

C. Distributed.Jobs

This namespace provides the majority of the API for users to write programs that conform to the system. Up to now most of the classes have been internal to the library, meaning that they cannot be directly accessed by any external assembly. Which is what any user program would be. All of the classes in the Jobs namespace however are publically available.

Jobs are the main building block of any distributed program run through the distributed system. A Job consists of some number of tasks. The Job class itself is an abstract class that must be implemented in the users code in order to be run through the system. The class contains the Schedule of the users job, the list of tasks to be performed and the list of result objects that will be generated and sent back to the user. The class is then split up into a number of methods some with default implementation and others that are abstract and must be implemented by the user themselves. RequestedNodes() is the method which tells the system how many nodes a program should be distributed too. By default this method returns one, overriding this method to set the desired number of nodes. It should be noted that selecting a number greater than the number of nodes available to the system will cause that program to be set to the total number of nodes available.

The Jobs default construct takes no arguments and initializes an empty results list along with setting the default schedule to fixed. Currently there are two types of schedules available to users, Fixed and Leapfrog. A fixed schedule means that tasks will be distributed to nodes in fixed blocks. For example take the case where a user requests four nodes and sets up twenty tasks. With a fixed schedule the first node would get the first five tasks the next node would get the next five and so on. With a leapfrog schedule the nodes get alternating tasks node one gets tasks 1, 1 + n, 1 + 2n . . . and so on. This case works well when the difficulty of tasks increases linearly, exponentially etc. rather than being a constant. If the problem increases in terms of difficulty of tasks a fixed schedule gives poor performance as the the runtime is directly dependent on how fast the slowest node finishes. This example demonstrates the need for load balancing among nodes. Since only the user is aware of the specific problem they are attempting to accomplish the task of correctly balancing the load falls to them. In theory the system could attempt to balance the load if one node finishes
and another one is still running but then you run the risk of duplicating efforts and it defeats the purpose of teaching programmers to correctly identify problematic problem sets.

The next required method that a user must implement when subclassing Job is the public abstract void Main(string[] args);. This is the method run by each node executing some number tasks. It should be thought of as an initializer of JobTask's. Since this method will be called by every node to setup the tasks to be run it is unwise to run computationally complex tasks in this method. Doing so will duplicate work on every node that is a part of your Job. Other methods provided by the Job class involve adding and starting tasks and adding and "compiling" results. Adding a task adds a section of work to the list of tasks for the nodes to perform. The StartTask(...) method should not be called by the user. This will be called by the node executing that task. The last feature that a Job provides is the ability to run a task at the end, once all the results have been generated, with the overloaddable method public virtual void RunFinalTask(). This method is called on the machine that submitted the job. This gives users the ability to do something with the results they have generated.

The JobResult class allows users to store results. JSON serialization is used under the hood to create a representation of the object to send across the network. Users subclass this object and then store these objects using the JobTask's AddResult(...) method. All results across all tasks are then compiled into one collection using the CompileResults(...) method.

Since the Job itself could not do the heavily lifting a task class was created called JobTask. A Job spins up one or more JobTask's to be executed on different nodes. Similar to how the Job works the only method that must be implemented in the JobTask is the public abstract void Main(string[] args);. This is the method that is executed by a node when StartTask(...) is called. This method should perform the majority of the work to be done by the system. Each task is guaranteed to only execute once. One final thing to say about a users program as a whole is that there need not be an entry point. In C# this means that there does not need to be a public static void Main(string[] args) method provided since it will not be called. This effectively makes the users program a class library and not an executable.

E. Distributed.Manager

All code for the server and tasks controlled by the server such as the scheduler reside in the Manager namespace. Communication between the manager and all other networked nodes is managed by a Proxy (See Distributed.Network). The manager itself keeps track of JobReference's which contain information about a job such as:

- Requested Nodes
- Username of submitter
- Path to the DLL
- User arguments to program

The Manager sits on the local IP address and waits for incoming connections. Each connection spawns off a new Proxy which manages the sending and receiving of data on that socket. The manager is responsible for keeping track of each node done through a reference to that node’s id.

The JobScheduler resides in this namespace as well. It is a part of the server but runs in its own thread. It waits for certain conditions to be true, namely that there is a job waiting and that the number of available nodes is less than or equal to the requested nodes from the job. A node is only available if it is not running a job. There is no notion of multiple jobs being run on the same node.

F. Distributed.Network

One of the most important classes in this system, the Proxy resides in the Network namespace. The Proxy is responsible for all communication between different parts of the system. The server, each node and any other connections to the server are managed by proxy objects. The proxy itself is made up of an AbstractReceiver and an AbstractSender which handle asynchronous reading and writing from the socket. Both are threads which communicate via the message passing and delegates. When the receiver receives a message on the socket it sends an OnDataReceived event. Anyone subscribed to that event via the delegate will get the message. Both the sender and receiver subscribe to the message so that if additional processing needs to be done on the receiving side it can be.

G. Distributed.Node

The final section is the Node. When it starts up it connects to the server, sends over information on what kind of resources it has available and then waits for Jobs. When a Job arrives it starts up a new process to execute that job and then communicates those results back to the server.

VI. System Startup

To create the system a command line application was built that performs all the functionality needed to setup the server, the nodes and provides the ability to submit a job to the system. The command line application is called defcore (Distributed Environment and Framework for .NET Core). To run the system execute the following command (show without options to produce the help message).
dotnet defcore.dll

Usage: defcore [options] [command]

Options:
  -? | -h | --help Show help information

Commands:
  load  loads a job to and executes it
  node  Runs the Node on a local IP address
  server Runs the NodeManager server on a local IP address
  submit submits a job to the system

Use "defcore [command] --help" for more information about a command.

Each argument then has its own list of options if more information is required. To start up the server the command would be dotnet defcore.dll server. It is important to note that the dotnet command line utilities are required to run any .NET Core program. This will output the IP Address that nodes should connect to. To run a node a similar command is used dotnet defcore.dll node <ip address> with the IP address of the server. Once some number of nodes have been connected to the server users can submit jobs with the command dotnet defcore.dll submit <job.dll> <args ...> where the job and arguments are the users program and the arguments to that program. The node that the job is submitted on must be the same as the one the server is running on so that the server can have access to DLL and does not need to first send the DLL to the server and then distributed it to the nodes.

VII. EXAMPLES

To illustrate how to use the system and to gather results a number of examples were created. Each example attempted to test some aspect of the system while being computationally challenging and relevant to areas within computer science.

A. Cryptography

Cryptographic problems and prime numbers are good candidates for parallel or distributed programs because of the complexity of the tasks that they involve. The first example involves Goldbach’s conjecture which states that every even integer greater than two can be expressed as the sum of two primes. Since it is not possible to write a program to verify this, that actually finishes, a modified problem state was created. Instead of every even integer a specific span of integers was picked. To make this computationally challenging the numbers chosen were 12 or more digits long. In order to achieve constant memory space the file first had to be split up into manageable chunks since reading a multiple GB file into memory was not possible.

1) Implementation: The implementation of the problem proved straight forward with the exception that to represent numbers large enough for the problem a BigInteger class is needed. C# provides such a class but the implementation is not as fully featured as other languages and so extensions to the class had to be made. The sequential program is given an input range where both the integers are even and the second one is larger than the first. Then starting with the first input for every even integer greater up until the end, we loop while true until we find the largest such integer that satisfies the condition mentioned above. To figure out if a number is prime an extension to the BigInteger class was implemented that uses the Rabin-Miller algorithm as seen in Applied Cryptography [3].

The Distributed version of this task involves creating tasks for different sections within the initial input range. An initial Job class is created that reads in the input from command line and sets up some number of Jobs to verify the conjecture and calculates the largest prime found for the summation. The JobTask itself is simply the sequential version of the program with modifications to only calculate a certain subrange. Lastly there is the JobResult which stores each node’s largest prime so that when all the results come back the largest of the largest can be calculated. This kind of example works particularly well in this distributed system because both the inputs and outputs are relatively small and are easily sent over the network. See the results section for specific timings.

B. Sorting

The next example involves sorting of large files in constant memory space. This example takes advantage of the fact that there is a shared file system for the computer science machines at R.I.T. This allows all executing nodes to have access to the same large file to sort. If the file to be tested had to be sent across the network most of the time spent would be in reading and writing the file and not in performing the task at hand. To generate a large enough file with enough entries in it numbers were randomly generated and then written to the file one number per line. Files tended to be around 25,000,000 lines long. In order to achieve constant memory space the file first had to be split up into manageable chunks since reading a multiple GB file into memory was not possible.

1) Implementation: Job in this instance, counts the number of lines in the file, and then partitions them to each JobTask. The individual tasks then break up the large file even further to fall within a certain memory range set for the example at 50MB. They sort the small chunks using the built in sort. At the end the final task for the node that submitted the job is to merge all the files back together. This kind of problem has a few prerequisites. First the file system you are working with needs to support reasonably large files of a couple gigabytes. Second all the nodes need to be run on a shared file system so that they can all have access to the
file being worked on. As examples go sorting is a classic computer science problem and works well in this instance because the file to be sorted cannot be read into memory all at once.

C. Buddhabrot

The final example showcases some of the new and upcoming features that are now possible with .NET Core. Cross platform image processing up until recently has been a challenge for .NET and C# because there were no libraries available and everything had to be done by hand forcing the programmer to have an in depth knowledge of how image file formats work. The library used for this example was ImageSharp [5]. Buddhabrot is a different way of rendering the Mandelbrot set. Instead of selecting initial points on the real-complex plane, one for each pixel, points are selected randomly. Each initial point is iterated using the standard Mandelbrot function in order to test whether it "escapes" from the region near the origin. Escaped pixels get reiterated, counting the selected pixels each iteration generates an image [4]. The term itself was coined "Buddhabrot" by Lori Gardi because of the way the image looks when rotated ninety degrees, the algorithm and technique themselves were developed by Melinda Green [4].

1) Implementation: For this example a few different iterations were implemented to demonstrate the sequential version, a parallel version and then a parallel and distributed version where each distributed node is also doing parallel computations. The sequential version works as follows, the main method sets up the constant variables being used such as image size number of iterations and number of samples. Samples in this problem are the number of times a random pixel is generated and then iterated using the Mandelbrot method. Then while the number of samples collected has not exceeded what was set, a random pixel is generated and then iterated to see whether it escapes and a count is made each time it does escape.

The parallel version of this realizes that each sample calculated is independent of all the other ones and thus samples can be calculated in parallel. To achieve this n arrays are created one for each thread and the bin counts are placed in this arrays. The arrays are then added up at the end.

The distributed version expands upon this idea by seeing that the parallel part can be run on multiple machines and there results can be added together and then reduced again by the main node. Writing out the image involves combining all the different arrays that were generated by the different threads and creating an image where each pixel is a value in the array. This creates the grayscale image seen in Figure 2.
The results show that as the number of cores exceeds four the speedup and efficiency of the program are diminished due to the fact that IO and network operations begin to dominate the runtime of the program itself. In general while the distributed version worked very well for certain cases, when it came to image processing a purely parallel version out performed the distributed due to the fact that the resulting images were large and time needed to be spent compiling together data. As a whole the distributed system performed well when working with small sized inputs and outputs even if the tasks themselves were computationally complex.

IX. CHALLENGES

There were a number of challenges revolving around working with .NET Core and setting up a distributed system. With new technology a lot of the times the features are not quite where they would be given another few years. .NET Core, when the project was started, went through a number of iterations. This meant that every new version had to be reinstalled on each machine in the CS department prior to testing. From a hardware perspective the resources to do this project existed but getting the proper access and disk quota turned out to be a challenge as the project required a good amount of physical space to store test data and results. In terms of a learning curve many iterations were needed before the system got to the point it is at today.

X. CONCLUSION & FUTURE WORK

As computing systems get larger the scale and complexity of problems we wish to solve grows accordingly. Often it is easier to find raw computing power than it is to understand how to split up a computationally challenging problem into more manageable tasks. With the distributed framework and library for C# this task becomes easier to manage by focusing on how to split up problems instead of how to send packets of data in a client server architecture. There are countless features to add to this system and the project as a whole has only scratched the surface. Adding an implementation of map reduce on top of this would be a good extension of the work as well as trying to get a shared memory model in place as opposed to the one currently used which is ridged and does not allow a back and forth between tasks and results.

XI. ACKNOWLEDGMENTS

Thanks to Professor Brown for taking on this project with me and teaching me C#. I had never written a line of C# before the semester started and now I feel like I know it better than other languages I’ve worked with for years. I would also like to acknowledge James Craig the system admin for the computer science department for being patient with me and upgrading the .NET Core version as well as providing me more disk space on the computer science machines.

REFERENCES