Winter 12-15-2018

Leveraging Grammars For OpenMP Development in Supercomputing Environments

Samuel Hunter
shunter@smu.edu

LEVERAGING GRAMMARS
FOR OPENMP DEVELOPMENT IN
SUPERCOMPUTING ENVIRONMENTS

Approved by:

Dr. Frank Coyle
Director: Software Engineering Program

Dr. Theodore W. Manikas
Clinical Professor

Dr. Jennifer Dworak
Associate Professor

LEVERAGING GRAMMARS FOR OPENMP DEVELOPMENT IN SUPERCOMPUTING ENVIRONMENTS

A Thesis Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science

with a

Major in Computer Science

by

Samuel W Hunter

B.S., Computer Science, Southern Methodist University

December 15, 2018

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my advisor, Dr. Coyle, for his guidance in the process of writing this thesis. Dr. Coyle, provided me with valuable direction and critical feedback that helped me shape my thesis into what it is today.

I would also like to acknowledge Beth Minton and Jim Dees for their support and promptness in helping me navigate the processes involved in making this thesis a reality.

Finally, I would like to thank Dr. Dworak and Dr. Manikas for joining my thesis committee and providing valuable feedback during the course of writing this thesis.

Hunter, Samuel W B.S., Computer Science, Southern Methodist University

Leveraging Grammars
For OpenMP Development in
Supercomputing Environments

Advisor: Dr. Frank Coyle
Master of Science degree conferred December 15, 2018
Thesis completed September 26, 2018

This thesis proposes a solution to streamline the process of using supercomputing resources on Southern Methodist University's ManeFrame II supercomputer. A large segment of the research community that uses ManeFrame II belong outside of the computer science department and the Lyle School of Engineering. While users know how to apply computation to their field, their knowledge does not necessarily extend to the suite of tools and operating system that are required to use ManeFrame II. To solve this, the thesis proposes an interface for those who have little knowledge of Linux and SLURM to be able to use the supercomputing resources that SMU's Center for Scientific Computation provides.

OpenMP is a compiler extension for C, C++ and Fortran that generates a binary using multithreading using in-code directives. With knowledge of OpenMP, researchers are already able to split their code into multiple threads of execution. However, because of the complexity of Linux and SLURM, using OpenMP with the supercomputer can be problematic. This thesis focuses on the user of ANTLR, a programming language recognition tool. This tool allows for the insertion of directives into code which serves to generate batch files that are compatible with the supercomputer scheduling software, SLURM. With the batch file, the user is then able to submit their code to the supercomputer.

Additional tools around this core piece of software facilitate a usable interface. In order to make the tool accessible to those without a background in software, the proposed forward-facing solution is a user interface to upload their code and returns a batch file that the user

can use to run their code. This eliminates the need for a new user to download, compile and run the ANTLR distribution to generate a batch file.

By abstracting away these complexities into a web interface, the solution can generate a batch submission file for the user. Additional tooling assists the user in finding empty nodes for code execution, testing the compilation of their code on the supercomputer and running a timed sample of their code to ensure that OpenMP is leading to a speedup in execution time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

This thesis is dedicated to all of my family's support and inspiration to push my boundaries further than I thought possible.

Chapter 1

Introduction

Ever-increasing computing power brings us new ways of solving problems that did not exist before. With the decline of Moore's Law and the advent of multiprocessors, a shift in computing strategy to meet the demand for new hardware arose that has come to affect every device that we interact with today.

As hardware technology advanced, there was also a need to advance the software that the hardware relied upon. Using multiple processing units added a challenge of managing instructions and the data shared across processing units. A shift from sequential programming on high clock speed units to parallel execution on multiple lower-clock-speed processing units marked a significant paradigm shift in programming.

In the 1990s, to ease the transition from sequential to parallel programming, the Architecture Review Board created OpenMP to create a set of directives that could be used in SMP architectures (Chapman, 2007). OpenMP then began to be included in C and C++ compilers. OpenMP enables a programmer to take a piece of sequential code, identify where it could be parallelized and then add a code-level directive specifying where to split the code into multiple threads.

With OpenMP as a tool, developers were able to parallelize their previously sequential code and parallelize code that would not be able to complete in a reasonable timeframe in its single threaded form. In the latter case, the application of a supercomputer can substantially add to the number of available threads for a program to use. Bridging the gap from OpenMP on a personal computing environment to a supercomputer is daunting, and the documentation is far sparser than OpenMP itself and is often specific to a particular supercomputer. A solution to this problem would allow researchers who wish to use a supercomputer with existing knowledge of OpenMP to adapt their program to be used be

used by the supercomputer without the explicit knowledge of its specific implementation details. This is the aim of the thesis.

1.1. Previous Work

Mack (2011) details Moores law as a result of push factors from the industry and pulling factors from applications of computing technology. In 1965 Moore predicted with minimal data that the number of components on a chip would approximately double each year. For many years this held true with a single processing unit. However, when this stopped holding true when more transistors could not fit onto a single chip without overheating the computing community was forced to reconsider their approach to assembling computer hardware. The economic as well as engineering limitations of expanding the processing power of a single chip lead to new thinking around computing resources.

Writing about efforts to improve computer in the 1960s, Smotherman (2016) recounts the design and fabrication of IBM ACS machine. In 1963, IBM began the development of Project X and Project Y which would eventually be used as a model for the IBM ACS line including the ACS-1 and the ACS-360. The goal at the outset was to increase the processing speed of the chips by two orders of magnitude over the existing IBM Stretch supercomputer. Neither of these claimed to create multithreading but laid the groundwork for branch pre-fetching and simultaneous multithreading. The inclusion of a second program counter within the ACS-360 was a response to the under-utilization of several functional units on the chip as a result of poor instruction level parallelism. Allowing the functional units to be better-utilized lead to increased instruction throughput. Although this project was eventually terminated, it for shadowed simultaneous multithreading.

Byrd and Holliday, (1995), discussed the early use of multithreading in hardware design and operating system implementation. In this context, multithreading was used as a way to address and take advantage of latency introduced by memory accesses. Increased processor efficiency is possible by executing multiple streams of independent instructions concurrently, called threads. During operations that would otherwise leave the CPU idle, another thread

can be acted upon by the processor. While this did not address the issue of latency, it increased the throughput of the processor.

At the time of writing, Byrd and Holliday note that the only commercially available multithreaded hardware was released in 1978. The Heterogeneous Element Processor (HEP), was manufactured by Deneclor Inc and did not succeed. However, multithreading continued in software implementations. Thread state, stored in the processors' registers and program counter, could be saved to the cache or main memory during interruptions and another thread could be loaded. This, however, introduced a thread context switching overhead making it inefficient to due frequently. Processor architecture designed for multithreading hoped to solve this problem by introducing additional sets of registers and program counters to decrease the context switch delay.

The interval at which the contexts are switched is broken down into two categories, coarse-grained and fine-grained multithreading. Coarse-grained multithreading refers to a strategy of executing a series of instructions from each thread before switching at an instruction that introduces latency. Fine-grained multithreading, like that implemented on the HEP, interleaves instructions from different threads in adjacent cycles.

Chou and Chung, (1995), detail the work that has been done in optimizing instruction execution for decades. Nearly all processors currently have superscalar processors which reorder instructions to push the number of instructions per cycle above one. Improvements beyond that attempt to predict branches in programs to speed up execution by speculating on the outcome of the branch and discarding the uncommitted and unused branch outcome. In these architectures, Ungerer writes, superscalar processors can issue instructions from different threads within a single cycle or in sequential cycles. The impact of this decision is seen in the usage of the processor horizontally and vertically. By using executing as many instructions per cycle as the processor can maximize its horizontal usage for a single cycle. By ensuring that there are a minimal number of empty cycles, a more egregiously inefficient usage of processor time, vertical usage of the processor is maximized. If the processor issues instructions from multiple threads in a single cycle, then it can compensate for a single

thread having low instruction level parallelism by filling the execution slots with instructions from a different thread.

To improve performance, speculative multithreading works by bringing the optimizations up a level from instructions to the thread level (Chou, 1995). Sohi and Roth (2001), outlined the use of speculative control-driven threads which do not require memory synchronization, allowing the order of memory instructions to be reordered correctly in the order of the threads. This is in contrast to non-speculative control threads that have no way of recovering from violations of data dependency meaning that any access to shared data must happen in sequential order.

A formalized model for speculative multithreading refers to a process that finds structures which have regular patterns that can be split from a single thread into multiple parallel ones (Liu et al., 2013). The SpMT execution model delegates the work of many different parts of a program to individual threads for execution while a single non-speculative thread is in charge of committing correct results to memory from the results of the speculative threads.

In 2016, Eckstein explored the origin of parallel computing as a means of overcoming the limitations of single-processor computers. Throughout the decade that followed 2000, there were diminishing returns associated with increasing the number of transistors on chips. This lead to manufacturers placing multiple cores on to single CPUs, enabling multiprocessing. This meant that multiple instructions could be executed at once on a single CPU.

Eckstein continued by explaining the different architectures that arose from the evolution of multiprocessing. The primary difference noted was the decision in how memory should be shared. In a shared memory system, all cores may have access to the same global memory. Systems that implement this are called symmetric multiprocessors (SMPs). At the other end of the spectrum, cores may have their own private memory, requiring messages to be explicitly sent and memory to be synchronized if any data is to be shared between cores.

With the rise of Symmetric Multiprocessing, there began an emphasis on writing code that could adapt to using multiple computing resources. Superscalar processing techniques along with speculative multithreading provided instruction and then some thread-level par-

allelism, however, code designed for parallelization removes the speculation from thread level execution ultimately optimizing the use of cores available. At a programming level, Eckstein (2016) notes two separate models, using either a data-parallel model or a control-parallel model. Data parallel models use a single master thread and split similar data manipulations across multiple processing units. Control-driven parallelism specifies a distinct control thread for each processing unit it wishes to use. This means that each processor will often take different paths through the program. In each model, there is the choice of passing messages or using shared memory to communicate data across threads. Eckstein (2016), concludes their discussion of programming models explaining that standards have been created to be used across manufacturers, including MPI for message passing and OpenMP for shared memory between threads.

Chapman (2007) describes the needs that surrounded sharing data amongst threads with the rise of multiprocessors. Manufacturers determined that in most programs a single processor depended on a value in memory created by a different processor. Because memory is shared between processors, this is not a networking or access problem but becomes a timing issue because processors may require a value only after it is produced by a different processor, something that is not guaranteed because clocks are not synchronized across cores. This gave way to manufacturers creating specifications in code, called directives, that dictated how threads would be created and the order in which shared data would be accessed. This made programming SMPs possible but meant that code written for one manufacturer might not work for an SMP created by a different manufacturer.

Chapman (2014) reviews the events that lead up to the creation of OpenMP. In the 1980s an informal industry group called the parallel computing forum began drafting a standard set of directives specifically for Fortran. However, the ANSI standard that arose from it was never widely adopted. The Architecture Review board took a step towards creating a standard in the 1990s with the advent of OpenMP. OpenMP was a cooperation between many stakeholders in the computing industry to establish a standard set of directives that could be used in popular compilers across many SMPs.

OpenMP enabled developers to bring their sequential programs into the multithreaded domain using in-code directives. The use of directives reduced the need for programs to be completely rewritten and made parallelization much more accessible to those with only a conceptual understanding.

The approach that OpenMP took towards parallel programming uses the shared-memory model. This means that the OpenMP package assumes that programs will be executed on processors that share at least some amount of memory. In order to execute instructions in a parallel fashion, OpenMP uses multiple threads. Each thread has access to the shared memory but may also have its own private memory during execution. This model is different from message passing which also enables parallel computing but requires threads of processes to explicitly exchange shared data that needs to be updated through messages instead of using the same memory.

Fosdick et al. (1996) details an emphasis placed on using computing resources to solve "grand challenges" before the advent of OpenMP. Director of the Office of Science and Technology Policy, William Grahm, released a five-year plan in 1987 to solve many of the fundamental challenges for which supercomputing could be used to advance research. As a result, the High-Performance Supercomputer Program was developed. The challenges selected includes ones which had a significant impact on science, economics, and politics. A few of those specified were the prediction of weather, climate and global change, semiconductor design and structural biology. The pattern in these problems was the desire to model a physical system in a computational environment to create a simulation of its behavior. While the HPCP program emphasized solving problems with new technology, the technology needed predated the five-year plan.

Supercomputers, a concept introduced in the 1960s, were a tool used to aid in the research of topics included in the High-Performance Scientific Computing Program. The benefit of supercomputers was their advanced logic elements and the parallelization they offered. The implementation of high-performance computers included two broad categories. The first one contains many individual processors that execute its own private set of instructions

asynchronously. These are referred to multiple instruction streams, multiple data streams (MIMD). In this variation, data can be sent between processors using a message passing paradigm or by sharing a buffer of memory. The second broad type of supercomputer architecture, processors operate synchronously. There is a single sequence of instructions executed by all processors on different sets of data. For this reason, the processors are referred to as "single instruction, multiple data" streams (SIMD). Supercomputers implementing a single instruction, multiple stream strategy use message passing to communicate between processors, as they execute the same instructions on privately held data. As OpenMP is a shared memory solution to parallelization, it is only able to take advantage of systems using a "multiple instruction, multiple data" stream architecture.

As mentioned previously, high-performance computing enables CPU intensive simulations to run within reasonable time bounds. Chen (2009) explores the use of commodity computer parts used in conjunction to enable simulations at the molecular levels in material science. Multiple hardware nodes can be used in conjunction to distribute loads of work across a high-performance network of processing units. Used in parallel, the combination of resources connected by a local network ultimately add another level to the use of SMPs, combining multiple processors which contain many cores. By connecting nodes, more memory and more computational power can be achieved at a much more reasonable price point than trying to scale a single processor.

Southern Methodist University unveiled its first supercomputer in 2014, what would become the predecessor to Mane Frame II (Abril, 2014). Mane Frame I was initially owned and operated in Hawaii by the U.S. Navy and contained over 1,000 separate nodes. Mane Frame II, released in 2017, reduced the overall number of nodes but increased the overall computing power. The second iteration organizes its nodes into queues which represent the different number of cores, different amounts of memory, different graphics card and different lengths of time for which they can be used for a single job. Each queue has multiple nodes and thus can handle multiple jobs at once. Every queue is managed by job scheduling software that the University has installed on the supercomputer.

1.2. Improving on Previous Work

Cluster computing enables a new paradigm in computing while also creating opportunities for the application of parallel computing to many different disciplines. Simulations and calculations that were once infeasible to accomplish in a reasonable time frame are now able to be completed in a fraction of the time. All that needs to be done is for the simulation to be written, parallelized and then run on a supercomputer.

While none of these steps is straightforward, the usage of supercomputers is not something that is taught in computer science education, let alone a pure science-based education. Furthermore, the number of resources for using a clustered computing environment is relatively limited as the audience for them is not comparable to that of scientific computing or parallelization. Moreover, multiple types of supercomputers use multiple types of operating systems which are running multiple variations of job schedulers.

The nature of the supercomputer is that understanding how to take advantage of it requires very domain-specific knowledge. For many researchers, this makes the effective use of a wealth of resources prohibitive. Making the supercomputer an accessible tool to students, researchers and professors removes a roadblock and possibly opens the door to projects that are now able to use the resources offered by the supercomputer properly.

This thesis focuses on bridging that gap by taking programs written in C, with OpenMP and generating the necessary batch file that will appropriately schedule the program to run on the supercomputer. Specifically, this thesis will use ManeFrame II, a supercomputer located at SMU as the target computing resource. ManeFrame II is a Linux based cluster supercomputer that uses the SLURM scheduler.

By bridging the gap between scientific computer and supercomputing, these resources can be better utilized, and with them, more simulations and computational-based research can be realized.

Chapter 2

Problem

Multithreading has implicitly made the study and development of code that works using multiple threads important. To harness the threading capabilities of new cores, users must write code in such a way that it takes advantage of the resources its running on. This problem was initially solved by vendors, but that made it challenging to write hardware-agnostic software. Put simply, moving code to a different machine was not possible. OpenMP was the solution to this problem taken by the Architecture Review Board, in the interest of vendors and developers alike.

Conceptually, multithreading is at the core of computer science and is a well documented technical topic as well as an extensively taught practical skill. Picking up multithreading, through language-built packages or from a compiler extension such as OpenMP is not difficult considering the breadth and depth of openly available resources.

This thesis focuses on the knowledge gap between creating a multithreaded program and running it on a supercomputer. This problem is relevant because it is relevant because it applies to anyone who wishes to begin taking advantage of the supercomputer. To some degree, there will be a conceptual hurdle for any new user of the supercomputer. Depending on their background this may be minimal and require reading through documentation about the supercomputer's job management software, SLURM, or it may involve learning an array of skills that need significant investment before one can see a speed up by using the supercomputer.

This problem exists because there is just not as much documentation about using supercomputers. Due to the cost prohibitive nature of supercomputers, they are a niche technology to which a relatively small number of people have access. The smaller community means that there is less of an audience for documentation and fewer people to ask and answer questions

about supercomputers. An additional factor that makes it more challenging to find useful documentation is the uniqueness of each supercomputer. Because supercomputers are not commodity products but instead built of many commodity products, none of them function precisely the same way. This level of uniqueness means that for each supercomputer there is commonly only a single point of reference.

Administrators of supercomputers do use abstractions at the software layer to hide the implementation of the hardware from the user, however even these software abstractions are niche to the supercomputer industry. By integrating batch job scheduling software into the supercomputer, administrators can offer the users an interface that has documentation beyond their own as well as a community that can provide guidance and instructions. Administrators of a supercomputer may mitigate the effectiveness of using a third party software by patching it to either optimize it for their supercomputer or to remove specific features. Furthermore, they may rarely update the third party software making it more difficult to find relevant documentation or solutions.

Even if the third party software is unpatched and updated by the supercomputer administrator, the community supporting is still small compared to that of OpenMP. As of June 24, 2018 Stack Overflow has 108,694 questions about multithreading, 4,258 questions about OpenMP and 442 questions about SLURM. While questions on Stack Overflow does not inherently mean that there is inadequate documentation released by the maintainer of SLURM, it does indicate that the amount of activity regarding SLURM is significantly lower than OpenMP.

To those who are not familiar with Linux or bash this problem is even more prevalent as they have to learn how to navigate a new operating system without a visual interface and learn how to use Bash to create their submission files. Applying computing resources to a topic of research can be a practical approach to a problem that can be modeled by a program. A thorough understanding of how that model is manifested in the program is crucial to generating an accurate data. As a result, understanding scientific computing is an essential skill for a researcher that would like to use programming in their study. However, the

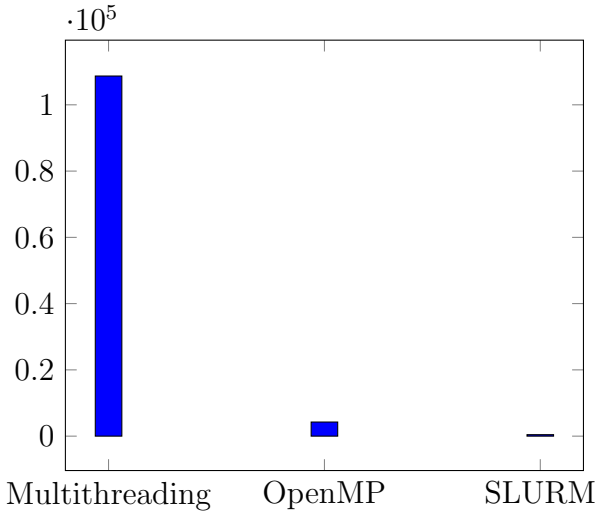

Figure 2.1. Frequency of Question Topics on Stack Overflow

supercomputer's interface requires understanding more than scientific computing. Using it then becomes a burden on the researcher to learn how to use a tool that is mostly irrelevant to the experiment itself. Providing a different, more guided interface to use the supercomputer could then encourage its use for those who know about scientific computing.

Chapter 3

Requirements

The proposed solution to alleviate the substantial learning curve is a tool to facilitate the use of a supercomputer for individuals not familiar with the details of high-performance computing. The primary aim of the solution is to reduce the amount of time needed to get a program running by assisting those with little knowledge of the technologies involved. As a result, the effectiveness of this tool can be directly measured by the velocity at which one can get up and running on the supercomputer. The solution does not attempt to provide a universal or comprehensive interface to the supercomputer, but instead to provide a straightforward path to seeing a program run on a super-computing node. Furthermore, because the audience of this tool does not have significant experience with software packages, the interface must be accessible and require minimal setup or preparation on the part of the end user. The tool should equip the user with the knowledge to reproduce the execution of their program without the assistance of the application. Ultimately the solution aims to provide enough context that the user may further pursue an understanding of super-computing-related software.

3.1. Scope

The scope of the proposed solution is to provide a way for people with knowledge of OpenMP and programming to use the supercomputer without having to learn Linux, bash, and SLURM before getting a job executed on the supercomputer. This solution assumes that users are experienced with programming in C or C++ and are familiar with OpenMP.

An assumption of knowledge of OpenMP is reasonable because harnessing the supercomputer for any sizable speedup will require executing code in parallel. Utilizing a single core on the supercomputer provides a negligible speedup over a home computer. To capture the

value of a supercomputer, it is necessary to know how to parallelize code. Users that want to take advantage of these resources must then understand the principles of parallelization through the process of multithreading. OpenMP is not the only way to achieve parallelization or to multithread code, but it requires the least amount of modification to sequential code to obtain it.

Unlike libraries such as Pthreads which are controlled in code, OpenMP only requires directives above existing sequential code to create threads. This approach removes thread-specific code and allows the compiler to decide how threads should be allocated across the available computing resources. The difference in implementation is highlighted in figures 3.1 and 3.2 which show us samples of code that use multithreading.

```
1 void mxv(int m, int n, double * restrict a, double * restrict b, double *
    restrict c) {
2       ...
3
4      #pragma omp parallel for default(none) shared(m, n, a, b, c) private(i, j
    )
5      for (i = 0; i < m; i++)
6      {
7          ...
8      } // end open mp
9  }
```

Figure 3.1. OpenMP Code Sample

To provide a practical realization of this proposed solution, we will focus on SMU's ManeFrame II supercomputer. Because supercomputers are not commodity hardware, they vary in implementation and resources. The problem this thesis addresses a broad audience of individuals who are interested in using a supercomputer, but the solution makes assumptions about how the supercomputer they have access to is implemented. The decisions made in pursuit of a solution are biased by the availability of particular supercomputer at Southern Methodist University. However, the software used to access the supercomputer, to schedule jobs, and to compile code are used across many supercomputers. This means that this proposed solution may be portable across other systems. However, the operating system and

```
int main () {
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
      ...
   }
   pthread_exit(NULL);
}
```

Figure 3.2. PThreads Code Sample

job management configuration will vary in ways that may cause misalignment in performance on different supercomputers.

The scope of the proposed solution is intentionally limited to the process of getting a user up and running on the supercomputer with a few steps that do not require extensive prerequisite knowledge. This solution does not intend to be an all-in-one solution. This is motivated by the fact there are many ways that a supercomputer can be effectively leveraged. The solution proposed only takes a narrow approach to using the supercomputer. This allows users to take the first steps without being overwhelmed with configuration options. Ultimately a researcher will need to enhance their understanding of SLURM to fine tune the use of their program on the supercomputer. They should not depend on this product for that.

3.2. Approach

The emphasis of this solution is on addressing two aspects of teaching the user how to use the supercomputer: demonstration and education. The first part of this approach is what requires the proposed solution to be a product rather than a set of instructions. By allowing the user to access and use the supercomputer through a friendly interface, it will enable them to begin conceptually understand the process in small increments. The user

can see the rewards of using the supercomputer and does not have to invest time up front to learn the intricacies of SLURM. The secondary goal of this solution is to break up the process of using the supercomputer into smaller, repeatable steps. Once the user can begin to establish a mental model of how the supercomputer job submission process works they will be able to adjust any part of the process to fit their use case more effectively.

The conceptualized solution taking shape as a product has three primary goals:

1. Generate a batch file for the user
2. Let the user see the program run
3. Show the user how to repeat the process

The first goal, generating a batch file, is a distinct one because it poses the largest conceptual hurdle to submitting a job for new users to SLURM. The batch file is a shell language file that allows the supercomputer decide where to execute the job and how to execute the job. An example batch file is shown in 3.3. To write a batch file, one must understand how SLURM works and the syntax to use in the files. This information is documented on their website but does not help new users get up and running without wading through a wealth of irrelevant information. Giving the user an initial batch file that will run their code lets them bypass the overload of information and supplies them with something that works.

The second goal, letting the user see the program run, walks the user through the remaining parts of using the supercomputer in a way that allows them to see the output from their program running successfully on the supercomputer. Another hurdle outside of creating the batch file in submitting a job is finding an open node. Helping the user find a partition or queue that has an open node means that the user does not have to queue their job and wait to see results. With an empty node, they can finally submit their first job to the supercomputer and have it run immediately. By the time the user has walked through the flow, they have taken each step that is needed to run a job on a supercomputing node.

The third goal, showing the user how to repeat the process, is important because the

```
#!/bin/bash
 #SBATCH  J samsfirstjob
 #SBATCH  o samsfirstjob_%j.out
 #SBATCH  t 120

 module purge
 module load gcc 6.3

 gcc first.c  fopenmp
 chmod +x first.c
 ./a.out
```

Figure 3.3. SLURM Batch File Sample

solution is not successful if users come to rely on it. As an educational tool, this product is one that provides a visual aspect to the process to assist in forming the mental model. Useful descriptions, in-code comments, and repeatable shell commands can bridge the gap between using this product and submitting jobs independently.

3.3. Functional Requirements

The proposed solution to the problem presented in chapter 2 is a product that should be usable for researchers and graduate students. As a result, this product needs to have a set of functional requirements that can be used as a baseline for success. Functional requirements detail the capabilities that the solution needs to address the problem previously presented adequately. Most of these are user-facing requirements but may also encompass qualities that are not externally visible [1]. Non-functional requirements are not listed for the sake of this thesis. Beyond having suitable processing times that allow a user to interact with the product intuitively, performance is not a first-priority issue for this project.

The following list outlines the fundamental functional requirements:

1. The application should explain to the user what kind of source file to upload
2. The application should accept a C or C++ source file
3. The application should validate that the uploaded file is a correct source file

4. The application should allow the user to enter a secure phrase to authenticate them with the website
5. The application should allow the user to click a button to upload their source file
6. The application should parse their source file and display a generate SLURM-compatible batch file
7. The application should notify the user if no OpenMP Pragma is found when parsing the source file
8. The application should allow the user to see which partitions on the supercomputer are currently available
9. The application should allow the user to select an empty partition and run their uploaded code on it with a minimal execution time limit
10. The application should notify the user before running their code if it fails to compile on the supercomputer
11. The application should notify the user when their job has been submitted, is processing and is complete
12. The application should time to running time of the user's program when it is compiled with OpenMP enabled and when it is compiled without OpenMP compiled
13. The application should allow the user to fetch the output from their program being run
14. The application should contain instructions that explain how to reproduce the job on the supercomputer without the application
15. The generated batch file should include comments explaining each line

3.4. Parsing For Pragmas

To generate a SLURM-compliant batch file using information from the user's source file a grammar needs to be applied against the source file itself. Constructing a grammar and then parsing the source file allows the product first to validate that there is an OpenMP pragma present as well as recognize any additional custom defined pragmas. The custom defined pragmas defined in the source file by the user are used by the product to generate corresponding code for the batch file. See table 3.1 for the available custom pragmas. However, if a user wants to get up and running with no specific configuration needs, there is a default batch file that will be returned. The default values will not be fine-tuned but comprehensive enough to work for most programs.

Table 3.1. Custom Pragmas

Pragma	**Description**	**Example**
NAME	Job name and the name of the output file	test˙job
TIME	Maximum job length in minutes	2
EMAIL	Email that job updates will be sent to	you@smu.edu
MEM	Amount of memory to be allocated for the job	64G

A language recognition tool called ANTLR will be used to construct the grammar and parse the source files. ANTLR is a full-featured tool that accepts a grammar and generates a lexer and parser that can be used against any text [2]. In this project, a grammar is developed specifically for recognition of pragmas. The generated parser then contains functions that can act on tokens defined in the grammar to generate the batch file. Figure 3.4 illustrates this process. This batch file is then ultimately returned to the user.

Taking the approach of using a grammar makes the product more flexible in the future. Because all of the code to generate the batch file is located in the parser, the grammar is left as a configuration file. Adding a new custom pragma requires adding code in only one place in the grammar. Furthermore, the power of the ANTLR parser can be extended to predict an optimal configuration for SLURM based upon the source code itself. By laying the groundwork for the scope of this thesis, the product is less work to extend in the future.

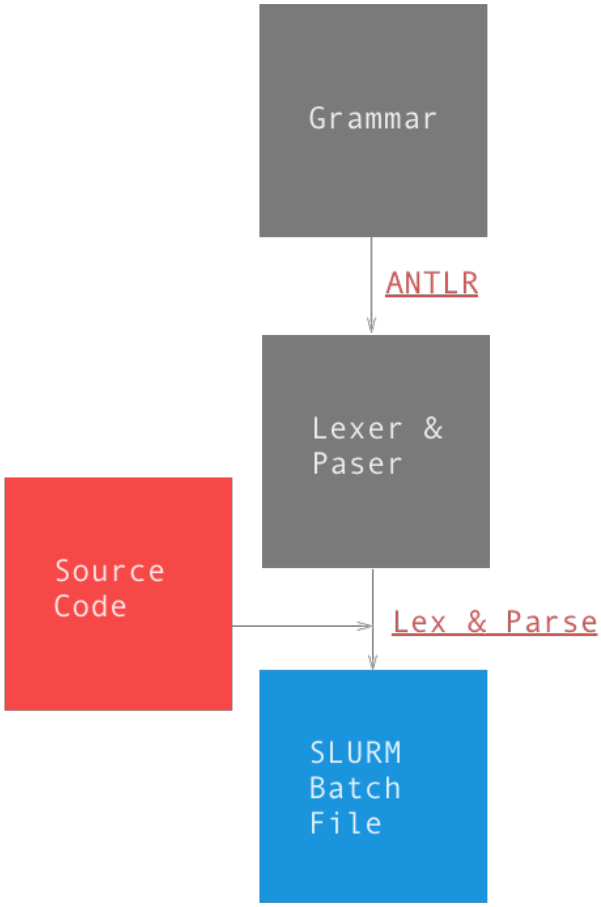

Figure 3.4. Generating a SLURM Batch File From Grammar With ANTLR

3.5. Determining A User Interface

Creating an intuitive interface for this solution is important as it needs to assist the user in understanding the supercomputer without getting in the way. The audience is the primary consideration is deciding on the appropriate interface for this product. The audience is primarily researchers and students who have some familiarity with programming but may not have an understanding of command line tools and installing packages on their computer. Ideally, the interface itself should not get in the way of the user or deter the user due to its complexity in usage or installation. The decision this thesis presents is to create a website on which to host the product.

Using a website allows the most substantial breadth of users to access the product in a

familiar interface environment. Furthermore, using a site does not require any installation so any computer that has a browser can access and use it.

3.6. Laying Out the User Interface

A significant focus of the product is creating a user interface on the web that is intuitive to use. The choice of using the web was motivated to remove any barrier to use, so the interface itself should provide that same benefit. A complicated interface will likely turn the user away from the solution and defeat its purpose. And while a complicated interface may enable more flexibility in the tools use, it will become burdensome for the first-time users. As a result, there is a focus on making a focused set of features that accomplish the functional requirements rather than a feature-rich tool that requires training or prior understanding to use.

To deliver a tool that is straightforward to use and makes an intimidating process seem manageable, this tool will break the process of executing a job on the ManeFrame into multiple discrete packages. Because the goal of this website is narrow, we do not want to present the user with many options upon initial load. To get the user into the flow of using the product, there will be a simple set of instructions and a short form that enables submission of a source file. Upon submission of their source file, the solution will validate that it is formatted correctly and contains an OpenMP pragma. Both of these checks will prevent the user from running into issues later in the flow and ensure that they have followed the instructions thus far. After successful execution, the application will return the generated batch file for display and then prompt the user to complete the next step.

An essential part of making the website intuitive is showing the user what they need as they need it. For example, upon initial load of the website the user needs to know what the website is and what the first step to using it is. As a result that is all that will appear to the user. Everything else is hidden initially, an example of what is shown to the user on immediately after the web page is loaded is laid out in figure 3.5. Only after the user has submitted a source file will any more content appear. When the batch file is displayed, it will

be laid out below the existing page elements. In addition to a batch file, the user will also be prompted to take the next step if they would like to. Once again, this provides a focused path for the user that prompts them to take to the step. By limiting the instructions and explanations to their bare minimum, it keeps their attention. More details, including steps to reproduce the flow, will be available if they click the help button at any point in the flow.

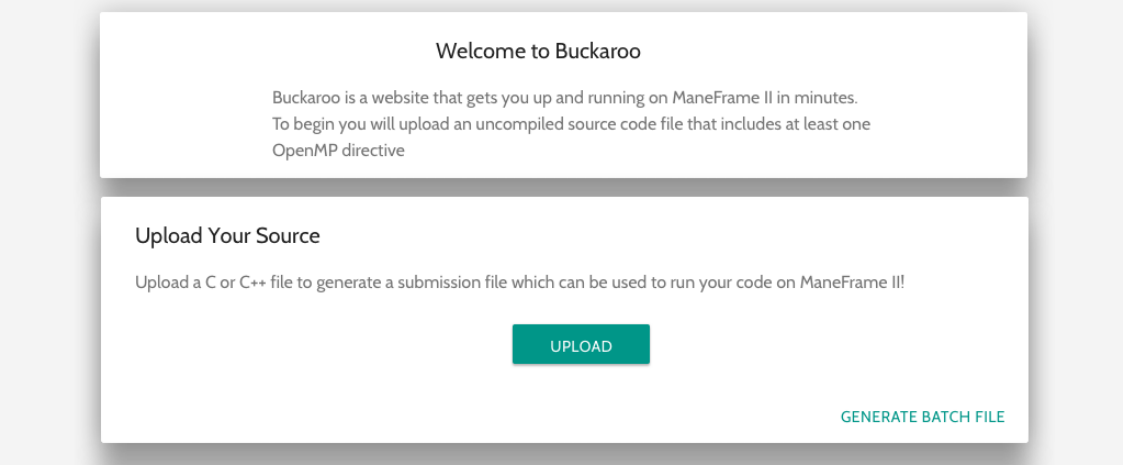

Figure 3.5. Initial Application Page Layout

The step-by-step flow of the user interface informs the user exactly what is expected to use the tool and breaks the process of submitting a job to a supercomputer into smaller chunks. Because one of the goals is to educate the user, breaking the process down into small steps that only become visible when they are necessary keeps them engaged. Furthermore, breaking the flow down into pieces forms discrete steps that when put together help form a mental model for individuals new to supercomputing. The manifestation of the discrete steps on the web page will be in the form of cards. Each card is an HTML element that is detached from one another and appears to float above the page. In this way, the layout accomplishes aids in effectively educating the user as well as walking them through the flow.

3.7. Educating The User

Guiding the user through the process of using a web application that is supposed to

educate them about using a supercomputer requires some amount of explanation. The goal of this tool is to minimize the number of instructions so that it is easy to follow and does not overwhelm the user. The instructions should give just enough context to complete the next steps without overwhelming the user with specifics that they will learn as they progress. Furthermore, having lengthy instructions will likely make the reader skim or entirely skip them in favor of trying to figure out the website on its own. To minimize the interface, the primary instructions should be on the page itself. These should be clear, definitive statements. The emphasis here is removing lots of the text that may distract the user from the functionality of the tool.

Supplementary instructions for use will also be included in the web application to provide context about what is happening while the user interacts with the tool. These instructions are less direct statements and are aimed to provide more details about how the supercomputer works and what the website is trying to achieve at each step. Because they are textual, they will be pulled out of the main page onto modals that will pop over the screen. Each step, represented on a card, will have an information button on the bottom, on the opposite side of the card as the call to trigger the next step. When the information button is pressed, a modal will appear over the page and will present more context about the purpose of the card it belongs to, as illustrated in figure 3.6. The amount of text will vary slightly but should be limited to keep the user engaged. The modal should provide enough context that the user can figure how to use the tool, more instructions about how to replicate these steps will be covered later.

For users who do not yet know what job they want to run on the supercomputer, a sample file will be provided. Any user that would like to try the website out without writing a program with OpenMP may use this file to get up and running. The file itself will contain comments explaining what it does as well as an OpenMP directive. The functionality of the file will be such that using OpenMP and a supercomputer with multiple cores will result in a speedup over running it without OpenMP enabled. For users that wish to work backward from a functioning example, this file will provide a boilerplate to which they can use to

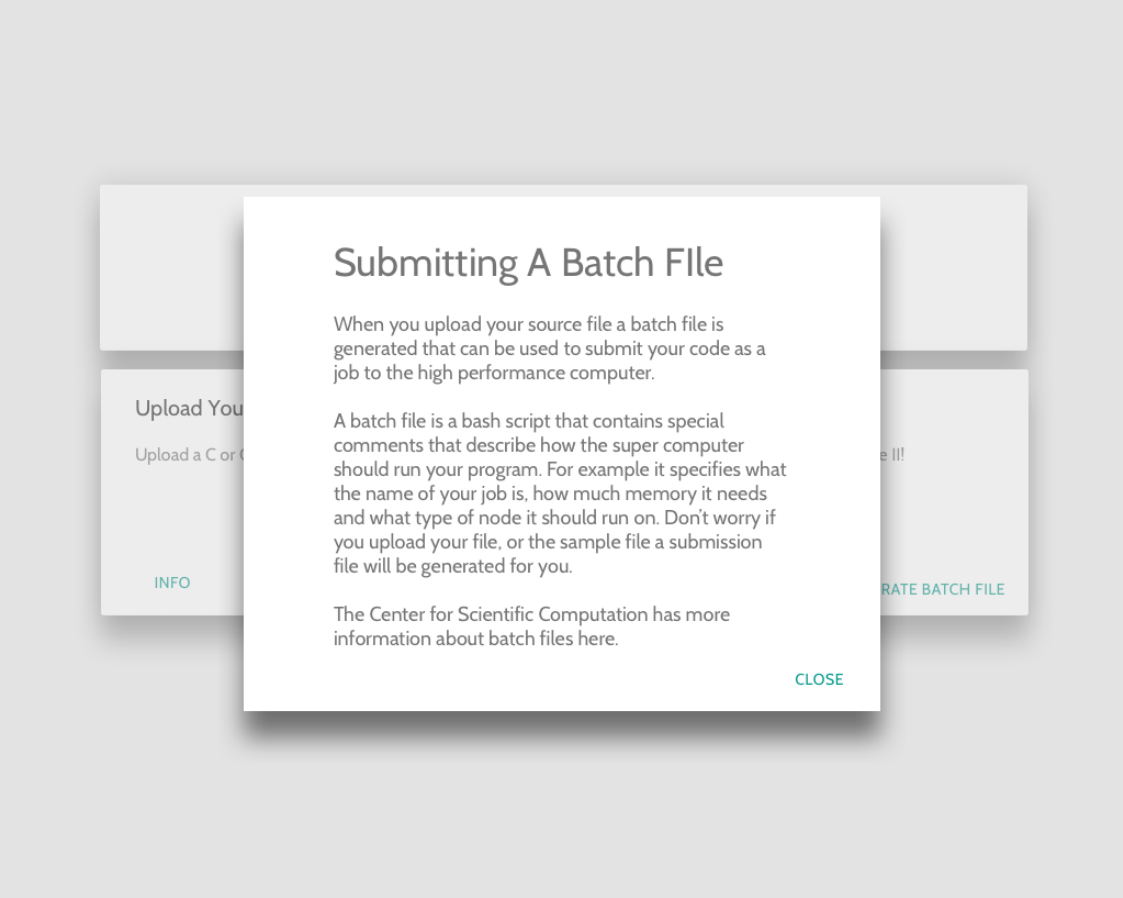

Figure 3.6. Contextual Information For First Step

adapt their idea or existing program. Ultimately this could also work well as an example file for people who want to use the site for demonstration purposes. The file will be hosted on Github so that it is publicly accessible and revisable. There will b a link to access the sample file in the first instruction modal on the website for users that do not already have a program to run.

Once the user has completed the flow and their program has run on the supercomputer they will be ready to take the next step and reproduce the results themselves on the supercomputer. They will still need to have some instructions to do this though since the website hides all of the interaction with the supercomputer. To provide instructions, a link will be included in an instruction modal to the Github repository that contains the code for the

website as well as a file that describes how to reproduce each step. By proving

Chapter 4

Implementation

4.1. Architecture

Implementing the proposed solution as it is laid out in chapter 3 requires creating a web application. The purpose of this section is to explain a possible architecture to achieve the goals detailed in this chapter. Because the realization of this solution is in the form of a website it will require using web-based technologies to implement. Specifically, it will require writing a web page or web application to which clients can connect. Furthermore, it will require interfacing with a web server or service that offers request handling to parse the source file and interact with the SMU supercomputer. The working name of this solution is SPUR, a name extracted from the word supercomputer and thematically appropriate for Southern Methodist University's ManeFrame II.

To fulfill these requirements, SPUR will adapt the client-server model. The client-server architecture model separates the interface that users interact with, the client, from the pieces of the system that perform the functions, the server [1]. From a development perspective, this allows for the separate development of the overall solution. It divides the portion that manages the data from the portion that displays the data in such a way that a user can understand and take action upon it. From a usage perspective, this model allows multiple users to access the same set of functions, data and computing resources at the same time. It also allows users to access their data from any device that is compatible with the client.

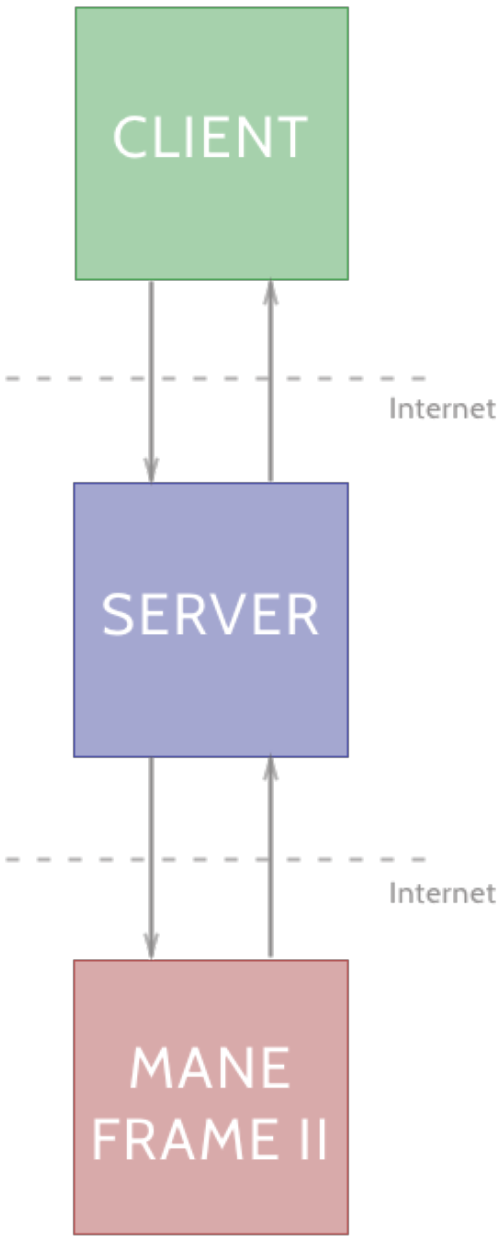

Figure 4.1. Three Tier Architecture Diagram

SPUR will take a less traditional approach to a common three-tiered architecture approach. In a traditional three-tier system a client would interact with a server which acts

as a middleman to a data store. The middleman would be in charge of authenticating the clients, authorizing the database transactions and doing any manipulation to the data before it enters the database or after it is retrieved from the database. Because a database adds extra complexity and minimal benefit for this solution, this will not be apart of SPUR. Instead, the third tier will be a login node on ManeFrame II. The login node will act as a data store for the files needed to run a job on the supercomputer as well as the data store for the files that are created while running the job. Furthermore, it will be the place from which the jobs are submitted to the supercomputer via the SLURM software package. To illustrate this relationship the three-tier architecture is presented in figure 4.1.

4.2. Implementation Details

The purpose of this section is to describe the technologies chosen to implement the proposed solution for the scope of this thesis. SPUR accomplishes the goals outlined in the functional requirements. While this section does not assume knowledge of all technologies described, it does a assume an understanding of the fundamental principles of network-based interaction between clients, specifically HTTP and WebSockets.

The motivation for choosing these technologies was to make this project extensible, readable, and most importantly functional. The bulk of this implementation is done in JavaScript-based languages and toolkits. This selection was made for several reasons. The first is familiarity and popularity. According to the 2018 Stack Overflow Developer Survey, JavaScript is the most popular programming language. For web-based technologies, JavaScript is the go-to language. By using it throughout this project, the code is more accessible to other people who want to either understand the code or extend it to add features, enhance features, or fix bugs. The second reason that JavaScript was chosen is partly a by-product of its popularity: there are libraries built to support many pieces of the infrastructure out of the box. The benefit of having a breadth of libraries to choose from is that the code written for this project can focus more on the business logic than the plumbing required to set up a website and server with which it can interact. Allowing the focus to be

on the features means that a more functional solution can be obtained in a shorter period than it may take using a different language.

Instead of taking a traditional approach to building a website which would generally use a lot of HTML, CSS, and JavaScript the author chose to build it in React. React is a library written, released and maintained by Facebook. Its benefits are briefly described on its website:

> React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when the data changes.

Instead of building out web pages inside of HTML files with CSS for styling, React builds interactive web pages from small, reusable pieces of code called components. A component defines the structure, styling, and functionality of a particular piece of the user interface. The benefit of these components is that they maintain separation of different pieces of the UI, they allow for the composition of other components from them, and they maintain their own state and define their own functionality.

```
class Square extends React.Component {
 render() {
   return (
     <button className="square" onClick={() => alert('click')}>
       {this.props.value}
     </button>
   );
 }
}
```

Figure 4.2. Example React Component

A short example of a component is displayed in figure 4.2. Some of this may look familiar, almost like HTML, this is called JSX. JSX is one way to define the structure of a component. It is a JavaScript module that looks like HTML for the sake of developer familiarity. Unlike HTML, JavaScript expressions can be embedded inside of JSX. Of course, JSX cannot run

natively in the browser. To get from components built with JavaScript and JSX to JavaScript that is run on the browser, React first transpiles all of the components into statements that look more like the code shown in figure 4.3.

```
const element = React.createElement(
  'button',
  {className: 'square', onClick: () => alert('click')},
  'Hello, world!'
);
```

Figure 4.3. Creating an Element In React

The createElement function in React will return an object that React then passes to its virtualized DOM. The virtualized DOM is an abstraction of the DOM that uses different semantics to update the real DOM. The most significant benefit is seen in an improvement in performance. Because updating the DOM is not a cheap operation, React will bundle updates to the real DOM in the virtual DOM to occur at the end of the event loop. The last thing before diving into the use of React is the use of React in a web environment is the most popular, but the flexibility of React allows the Virtual DOM semantics to be applied to any user interface such as a smartphone, a smartwatch or virtual reality headsets.

For this project, the author chose React because it made building the website quicker and more extensible in the future. Because this website is relatively minimal in its interactions and feature set compared to many consumer-facing web apps, it may seem that adopting a heavy-handed framework like React is excessive. However, using React has significantly increased the velocity at which features can be written as a result of the composable nature of components and the exhaustive number of packages that can be used with React to create attractive interfaces. Furthermore, because the components are easy to edit, create and remove, extending this web interface will be straightforward for a new developer as opposed to editing a large or several large HTML files with elements that depend on one another.

To expedite the process of building this application, the implementation of SPUR uses resources from Material UI, a React library that has pre-built components. Material UI is

a free and open source and has components that are built to align with Google's material design pattern. These components roughly match the designs already created and save a significant amount of time spent on styling specific elements. Building with Material UI makes it a lot easier to focus on the layout and less on the minute details of a button or cards styles.

The server code to handle the client's request is also written entirely in JavaScript. More specifically the server's request handling code is written in Node.js, a runtime built on top of a JavaScript engine so that code written in JavaScript can be executed as a script on a host machine. It is important to note that the server request handling at the network layer will be done by Nginx, an alternative to Apache's web server, but these requests will then be passed to the Node.js script to interpret and take action on each request.

The Node.js runtime supports creating a server that listens for HTTP requests on a given port, which can then be handled. However, on top of Node.js, several additional frameworks handle server request handling. Frameworks add support for many infrastructure concerns that Node.js does not accomplish on its own. Frameworks also add middleware, code that is run against each request to manipulate or route it, that makes developing server code more straightforward and more feature-focused than infrastructure-focused.

To provide a greater focus on the application code, SPUR is implemented with Hapi, a Node.js framework. Hapi is a free open source software module written in Node.js that satisfies many of the requirements of initially setting up a web server out of the box. Figure 4.4 demonstrates the relationship between the host machine, Nginx, Node.js, and Hapi. The two assumptions that are made on this diagram are that the host machine has port 443 accessible to the Internet and that the Hapi server instance is run on port 3000. Requests from the client can be sent to the server through port 443, the port associated with HTTPS, which will then be routed internally by Nginx to port 3000 where a Hapi server is listening for requests via the Node.js runtime. Because port 443 is being used, Ngnix will present a certificate that can validate the SSL claim and verify that the connection to the host is secure.

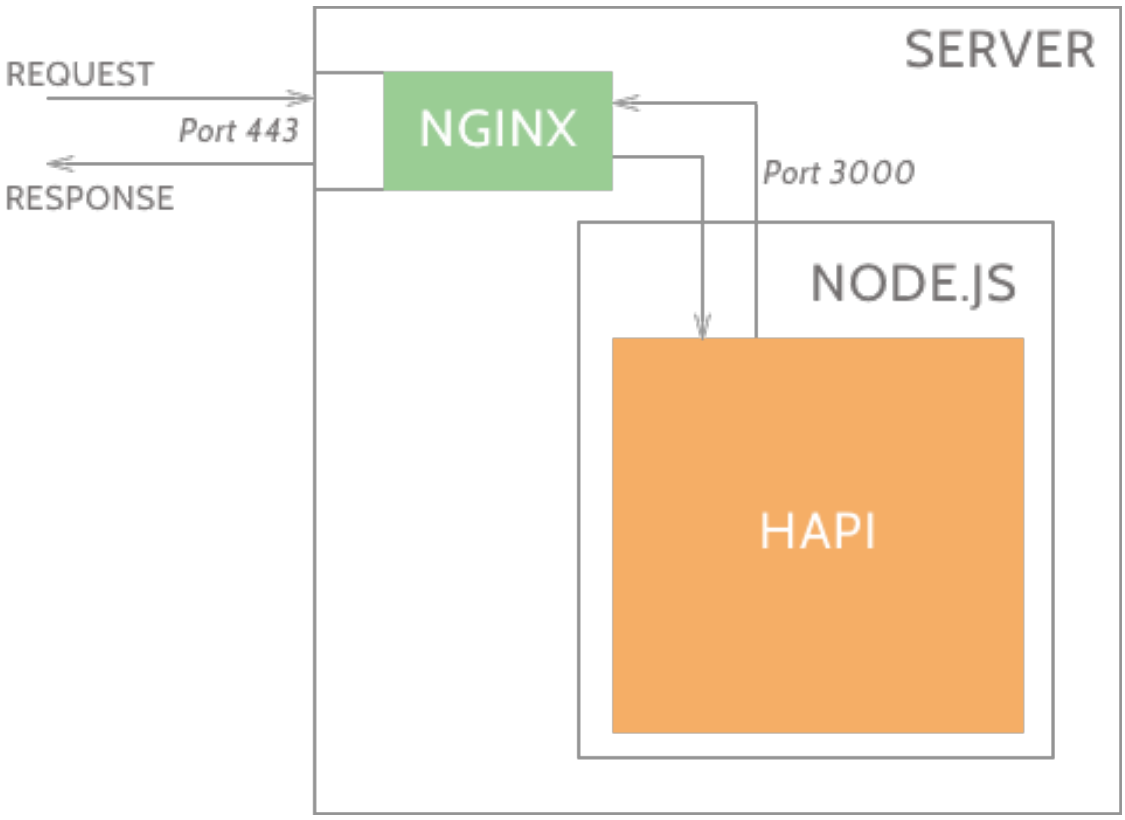

Figure 4.4. Server Request Flow

Hapi will then process the request it has received from Nginx. The Hapi software package comes with routing out of the box, such that it can read the destination URL of the request and forward the request to the correct handler function. Furthermore, in that process, Hapi will construct a JavaScript object that contains all of the information about the request. The handler is then able to read any query parameters in the URL without having to parse them or fetch any payload that is associated without needing to check its type and then convert it to a JavaScript object. The code in the handlers needs to manipulate, store, or otherwise act upon the request and then return an appropriate response to the client. Once the response has been sent from a Hapi route listener, it will be sent back to Nginx and routed to the client that issued the request.

To accomplish the functional requirements set forth it section 3.3, the following routes will be created:

1. Home: this route is in charge of simply returning the bundled and minified HTML and CSS from the React application so the client's browser can render it.

2. Upload: this route is called when a user initially uploads a source file, it is in charge of saving the file on the server for later use, generating a batch file and returning the batch file to the user.

3. Partitions: this route is called when a user clicks on the button to find available partitions on which to run their source code. This route will issue a request to ManeFrame II and then return a formatted list of partitions that currently have idle nodes.

4. Run: this route is called when a user clicks on the run job button after selecting a partition. It is in charge of regenerating a batch file to submit with the job, fetching the saved source code file, transferring both files to Mane Frame II and then submitting the job via SLURM on the supercomputer. It then returns a message indicating the success or failure of the request.

5. Fetch Output: this route is called when the user has clicked the fetch output button after their job has finished running. This route is in charge of fetching the resulting output file from the ManeFrame II login node to return to the client for display.

6. Receive Mail: this route is called when the job has begun running or has completed and emitted a message to the client to indicate the state change. This route and its purpose will be discussed later in this section.

Because jobs on the supercomputer are long-running, their success or failure cannot be encapsulated in the scope of one request, response pair. That is, once a job is submitted to ManeFrame II it may take anywhere from 10 - 120 seconds to complete. This is well beyond the expected duration of a single request. As a result, the server will need some way to know when a job is finished so it can inform the client and the user when the job output is available. This requirement does not conform to the request, response nature of the HTTP protocol as the client will not know when the job is complete and thus when to fetch the job

output. Instead, the server will need to be notified when the job is completed to inform the client about the job's completion then.

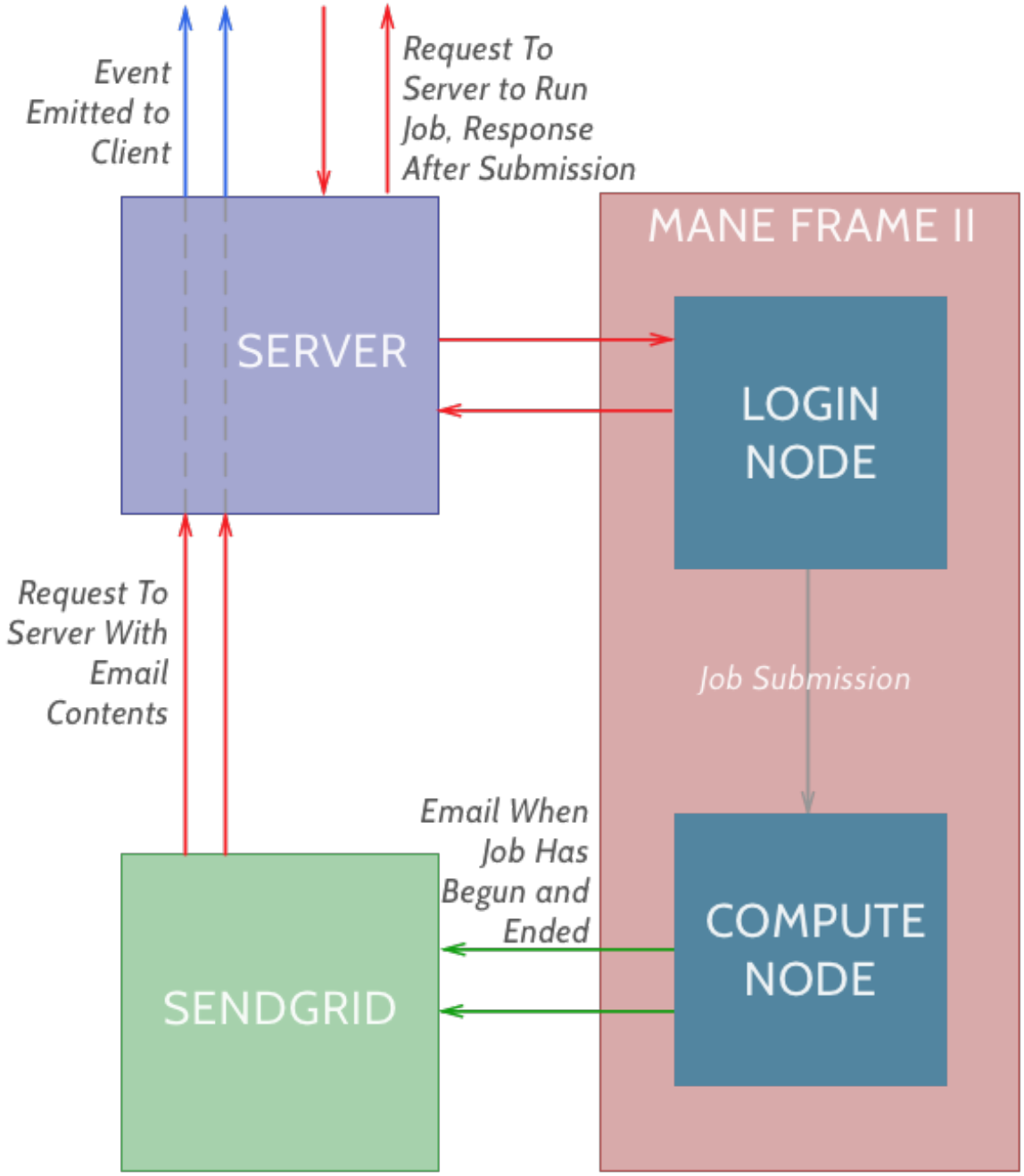

Figure 4.5. Supercomputer Job Notification Flow

The process of notifying the client of a job's completion requires several steps. Figure 4.5 outlines the components involved this flow. The first part of this flow occurs when the user clicks the button to run the job on the website. This triggers the request to submit the job to the supercomputer. If the job submission is successful, then a response indicating such is returned to the client. Shortly after this interaction, assuming the partition specified by the batch file indeed has an empty node, the job will begin running on a computational node within the supercomputer. When this happens, an email is sent from the supercomputer, via SLURM indicating that the job has begun being processed. The email address, specified in the batch file is configured through DNS to have all of its mail directed to Sendgrid.

Sendgrid is a platform typically used for companies to sent marketing and transactional emails. One of the features they offer is being able to receive inbound emails and then send an HTTP request with the email data and metadata to a specified URL. In this case, the URL that the HTTP request is sent to is the server's receive mail route outlined earlier in this section. When the request is received by Hapi, it will then be able to notify the correct client based on the job id in the email about the job's status. To notify the user, the server emits data to the client via WebSockets.

WebSockets are another form of networked communication that allows a client and a server to communicate outside of the bounds of the standard HTTP request, response paradigm. WebSockets enable the server to send messages to the client without first receiving a request from the client. Without WebSockets, if the server wanted to send a message to the client, the client would have to continually ask the server, called polling, for any new messages. Using WebSockets, a client can connect to a server and then subscribe to a specific topic. Then the server can publish a message to the same topic to which the client is subscribed. The client will then receive this message and can act upon it. For the proposed solution the client will subscribe to a topic using the job id that it receives when the job is first submitted. Subsequently, when the job begins processing and is finished processing the server will publish a message to the topic using the job id as the name. The client can then update the browser to reflect the change in the job's state.

Once the client knows that the job has been completed, it will allow the user to fetch the output file. When the client issues a request to the server's fetch output route, the server will go to the Mane Frame II login node and fetch the output from the job. The server will then return the contents of the file, and the client will display them on a card. At this point, the flow for the tool will be complete.

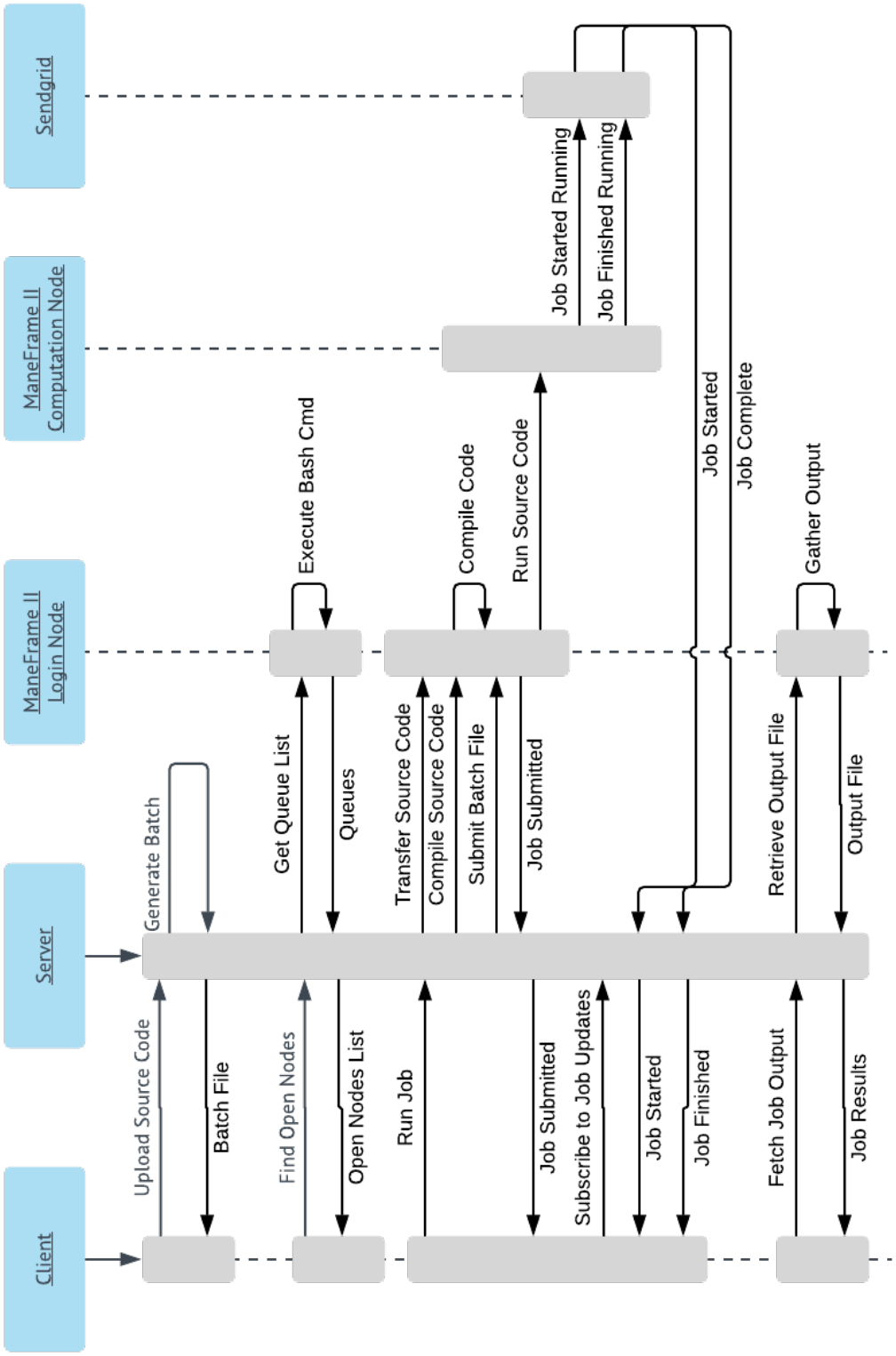

Figure 4.6. Application Sequence Diagram

4.3. Sequence of Interactions

To understand the proposed solution altogether, we now put together all of the pieces discussed in section 4.2. Figure 4.6 brings together all of the moving parts involved with SPUR. The structure of the figure is a sequence diagram. The purpose of this diagram is to describe how each technology is used throughout the tool's interaction sequence. Furthermore, it maps the interaction to the technology involved.

The first part of this sequence corresponds to the first interaction the user has with the website, uploading their source code. This triggers a request from the client to the server via the upload endpoint that has been written in Hapi. The source code file is then saved on the server, and the contents are loaded so it can then be passed to ANTLR to parse. Once ANTLR has parsed the code, the relevant pragmas are returned to the route handler which are then used as input to the generator function which yields the text for a SLURM batch file. Assuming there are no errors in parsing the source code or generating the batch file, the batch file text is then returned as the response to the original HTTP request. Upon successful receipt of the response from the server, the client will display a new card with the batch file text on it. This completes the first part of the interaction which corresponds to the first grey rectangle see under the client heading.

The second part of the sequence involves finding open partitions on which the user can run their source code and generated batch file. This interaction begins when the user clicks on the find open partitions button. The client sends a request to the partitions endpoint. The handler function then uses ssh to interact with the Mane Frame II login node and query for partitions with idle nodes. Once that data is received, it is parsed from its string format to a map-like data structure. The data is then returned via the HTTP response to the client. The client then displays a new card containing a table with a row for each partition that has an idle node. Part of each row is selectable as it is used for input into the next interaction.

The third interaction is the heaviest and involves each piece of technology discussed in section 4.2. When the user clicks the run job button a request is sent to the run route on the server. The first thing the server does is look for the source code that the user

had previously uploaded. If it cannot find it an error is returned to the client. If there is no error, the handler function then proceeds to regenerate a batch file to submit to the supercomputer alongside the source code. This batch file differs from the batch file that was generated previously because it injects a different, application specific email address that directs all emails notifications to Sendgrid. The batch file submitted along with the source code also specifies the partition selected in the previous step of the application flow and includes a maximum time of two minutes to limit the time spent on the job. The limit is placed so that the user can get a result within a reasonable amount of time. The server's handler function will then transfer the batch file and the source code to a login node on Mane Frame II. As a final precaution before submitting the job, the route will tell the supercomputer to compile the source code. If source code compilation fails, an error is returned to the client informing them of the compilation issue. Otherwise, the run route will issue a command to Mane Frame II to submit the job to be executed. This command will return the newly submitted job's id. The job id is then returned to the client via an HTTP response. At this point, the client will display a new card that shows the status of the job.

After the client has displayed the job status card, it will subscribe to a WebSockets topic where the name is the job id that was returned from the server. Once this is done, it will be ready to receive any job updates. In the meantime, the job should start running on a supercomputer computation node. When the job begins and ends, SLURM will send an email to address specified in the batch file. In this case, Sendgrid will receive this email and issue an HTTP request to the server's receive mail route with the contents of the email. The receive mail route is in charge of parsing the job id and the job's status from the email content sent to it from Sendgrid. If the route is able to parse out a job id and status successfully, it will then emit an event via WebSockets with the job id as the topic and the new status of the job as the event body. Each time an event message is received by the client, it will update the user interface to reflect the status of the job. Specifically, when an event message is received indicating that the job has completed, the client will enable a button on the job status card to fetch the results on the job.

The last interaction is kicked off by the user clicking the fetch results button on the job status card. Upon clicking the fetch results button, the client sends a request to the server's fetch output route. The handler function for the route then issues a request to the Mane Frame II login node to fetch the contents of the job's output file. The job's output file is then returned to the client at which point the client displays it on a final card. The output contains the runtime for the source code when it is compiled with OpenMP as well as the runtime for the source code when it is compiled without OpenMP.

4.4. Deployment

To allow users actually to interact with the tool, it must be deployed in such a way that browsers connected to the Internet can access it. While there are many ways to accomplish this, the solution used for this tool is Amazon's web service suite, AWS. Specifically for SPUR, an elastic compute instance has been instantiated. The instance itself is called a t2.micro which means that it is allocated 1 virtual CPU and 1 gigabyte of memory, enough power to host the server and do any necessary computations. To turn the instance into an accessible server, Nginx is installed and will handle all requests to port 80 and port 443, the ports that correspond with HTTP and HTTPS respectively. These requests will be forwarded to port 3000 where the Node.js and Hapi process will be running indefinitely, ready to receive and handle requests.

The client code is hosted on the same AWS elastic compute instance as well for simplicity. Before starting the server script, the React code is built and bundled for production on the server. When the Node.js process has begun, and a browser attempts to connect to it, the server will then return the bundled client code for the browser to display. Any requests originating from the client are then sent back to the server using SSL to secure the connection.

The benefit of this hosting arrangement is that it requires no equipment to host the website, it requires no set up on behalf of the user and the computing resources needed are minimal enough to be cost-free from Amazon Web Services.

Chapter 5

Summary

The focus of this thesis is to lower the barrier of entry for the high-performance computing resources available to students and faculty at Southern Methodist University. ManeFrame II offers over 11,000 cores across 306 nodes, and has enabled research in biology, chemistry, civil engineering, and many other fields. Still, the supercomputer is not very accessible to newcomers. The potential for this hardware is only bound by the computational curiosity and ingenuity of its users. Unblocking users from the technical layers of complexity, some niche to the supercomputer itself, allow energy and research time to be devoted to running the simulations and experiments on ManeFrame II instead of the software that is used to manage the supercomputer.

The result of this project and the implemented solution to the problem is a web application called SPUR. SPUR enables students and researchers at SMU to learn about ManeFrame as well as run jobs on the supercomputer without leaving the website. The goal with SPUR is to create an application to act as a user's first touch-point with ManeFrame. Using SPUR facilitates, with its user interface, the process of submitting a job on the supercomputer. Additionally, it enumerates the steps necessary to run a job without using SPUR so users may eventually submit jobs to the supercomputer themselves. SPUR makes the initial interaction with the supercomputer as fluid and straightforward as possible. A minimal UI guides the user through a step-by-step flow that breaks the otherwise elaborate process of submitting a job on the supercomputer into digestible and repeatable pieces.

For those users that have an understanding of scientific computing and a background in programming with a conceptual understanding of multithreading, code that has already been written using OpenMP can be translated to adapt to the supercomputer. A language recognition tool, ANTLR, is used to recognize special comments in the user's source code

and translate them into directives for the job scheduling software. SPUR is built with the intent of addressing an audience that has their code running in multiple threads on their own machine but would like to run it on the supercomputer. Additional tooling enables the user to retrieve the necessary configuration file, find an available node to run the job on, submit the job to run on the supercomputer and retrieve the output from the supercomputer.

The benefit of building a web-based solution to this problem is its accessibility and maintainability in the future. The technologies used to build the web app are accessible, well documented, and entirely written in JavaScript, minimizing the skill required to maintain the app. As users are exposed to the app and have questions and suggestions, the web app can easily be improved and updated. This aspect of the solution turns a thesis project into a product that is usable after the thesis is complete.

SPUR opens up these resources to a much larger audience by reducing the amount of prerequisite knowledge necessary to use the ManeFrame II. Enabling a wider audience for SPUR leads to more projects can benefit from ManeFrame II usage. Moreover, the Center for Scientific Computing will have a greater incentive to continue ManeFrame's growth as it can build a case for more funding from SMU. Ultimately though, this tool will prove beneficial if it can draw people from fields outside of computer science who are otherwise blocked by the knowledge barrier to entry to use the supercomputer successfully.

BIBLIOGRAPHY

[1] R. Stephens, *Beginning Software Engineering*. John Wiley & Sons Inc, 2015.

[2] T. Parr, *The Definitive ANTLR 4 Reference*. O'Reilly UK Ltd., 2015.

[3] "Center for Scientific Computation - CSC."

[4] Sendgrid, "Setting up the inbound parse webhook."

[5] K. Kirkpatrick, "Parallel computational thinking," *Communications of the ACM*, vol. 60, pp. 17–19, nov 2017.

[6] P. J. Denning, "Remaining trouble spots with computational thinking," *Communications of the ACM*, vol. 60, pp. 33–39, may 2017.

[7] C. . S. B. . G. M. Porter, Leo ; Lee, "Education: Preparing tomorrow's faculty to address challenges in teaching computer science," *Association for Computing Machinery*, vol. 60, p. 25, May 2017.

[8] J. Ortiz-Ubarri, R. Arce-Nazario, and E. Orozco, "Modules to Teach Parallel and Distributed Computing Using MPI for Python and Disco," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, may 2016.

[9] M. K. Smotherman, E. H. Sussenguth, and R. J. Robelen, "The IBM ACS Project," *IEEE Annals of the History of Computing*, vol. 38, pp. 60–74, jan 2016.

[10] T. Abel, *ReactJS: Become a professional in web app development (Javascript Frameworks) (Volume 3)*. CreateSpace Independent Publishing Platform, 2016.

[11] *Topics in Parallel and Distributed Computing*. Elsevier Science, 2015.

[12] Architecture Review Board, *OpenMP Application Programming Interface*, version 4.5 november 2015 ed., 2015.

[13] Danielle Abril, "The ManeFrame: SMU's new system raises Dallas' supercomputer capabilities," *Dallas Business Journal*, 2014.

[14] C. Ferner, B. Wilkinson, and B. Heath, "Using patterns to teach parallel computing," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IEEE, may 2014.

[15] A. Ortiz, "Server-side web development with JavaScript and node.js," in *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*, ACM Press, 2014.

[16] B. Liu, Y. Zhao, Y. Li, Y. Sun, and B. Feng, "A thread partitioning approach for speculative multithreading," *The Journal of Supercomputing*, vol. 67, pp. 778–805, aug 2013.

[17] V. G. Cerf, "Computer science education—revisited," *Communications of the ACM*, vol. 56, p. 7, aug 2013.

[18] C. A. Mack, "Fifty Years of Moore's Law," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, pp. 202–207, may 2011.

[19] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using MPI and OpenMP on multi-core parallel systems," *Parallel Computing*, vol. 37, pp. 562–575, sep 2011.

[20] M.-H. Chen and T.-L. Li, "Construction of a high-performance computing cluster: A curriculum for engineering and science students," *Computer Applications in Engineering Education*, vol. 19, pp. 678–684, may 2009.

[21] B. Chapman, *Using OpenMP*. MIT University Press Group Ltd, 2007.

[22] G. S. Sohi and A. Roth, "speculative multithreaded processors," *computer*, vol. 34, pp. 66–73, apr 2001.

[23] *Encyclopedia of Operations Research and Management Science*. Springer, 2001.

[24] L. D. Fosdick, E. R. Jessup, C. J. C. Schauble, and G. Domik, *Introduction to High-Performance Scientific Computing (Scientific and Engineering Computation)*. The MIT Press, 1996.

[25] H.-C. Chou and C.-P. Chung, "An optimal instruction scheduler for superscalar processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 303–313, mar 1995.