UNIVERSITY OF HERTFORDSHIRE

School of Computer Science

7WCM0031 Software Engineering MSc Project (Online)

Final Report

September 2016

Distributed Computation without Message-Passing: Remote Function Invocation at a System Level

L. K. Manickum

Abstract

Remote function execution is the ability of a running program to execute a selected function on a remote computer. This thesis examines remote function execution in detail, starting with an implementation of remote function execution for Linux *systems, moving on to the generation of a performance profile for the implementation measured against an existing concurrent system framework. A few of the more interesting possibilities using this new technique are briefly explored and an implementation of a domain-specific language which uses the remote function implementation to automatically parallelise the evaluation of arithmetic expressions is developed. The important contributions of this thesis are:*

- *A functional framework and library to enable remote function execution,*
- *A demonstration executing unmodified binary programs on a cluster of nodes,*
- *A performance profile of remote function execution as implemented by the software,*
- *A domain-specific language leveraging the computational power of a cluster for solving expressions in parallel,*
- *The creation of a new parallel-processing model which uses neither* shared-memory *nor* message-passing.

Further interesting Research Questions are developed in the findings, possibly with a lead into future avenues of research into remote function execution.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr Thiago Matos Pinto for his guidance throughout this thesis, and the module leaders of the MSc programme at the University of Hertfordshire for their support and help.

I also thank my employer, DetNet SA, for their support and help, and special thanks to Christo Botha for help with creating the Performance Testing Environment that was used in this project.

I would also like to thank my wife Valerie for her strength and support throughout the entirety of this programme.

Contents

1 Introduction **2**
1.1 Overview 2
1.2 Proposed Mechanism 3
1.3 Research Questions 3
1.4 Process 4

2 Multiprocessing: Current State of the Art **6**

3 Core Objective: *Rexel* Implementation **9**
3.1 Goals 9
3.2 Requirements 10
3.3 Overview of Concept 10
3.4 Architecture and Execution 11
3.5 Challenges 12
3.6 Summary 14

4 Core Objective: Comparison of *Rexel* and *MPI* **15**
4.1 Introduction 15
4.2 Executable Programs 15
4.3 Methodology 18
4.4 Test Setup and Architecture 18
4.5 Capturing and Preparation of Data 20
4.6 Analysis of Results 22
4.7 Outlier Discovery 24
4.8 Summary 26

5 Advanced Objective: Porting Unmodified Programs **29**
5.1 Overview of Objective 29
5.2 Implementation Details 30
5.3 Result of Experiment 31

6 Advanced Objective: Partial Shared-memory Processing **33**
6.1 Language Grammar 34
6.2 Expression Evaluation 36
6.3 Shared and Non-shared Memory 38
6.4 Summary 40

7 Findings **43**
7.1 Objectives 43
7.2 Research Questions 44
7.3 Further Avenues of Research 46

APPENDICES **50**

A Format of *Rexel* Configuration **51**

B The Uncommon Lisp Environment **53**
B.1 Expression Evaluation 53
B.2 Design of Expression Evaluator 55
B.2.1 Evaluation 55
B.2.2 Variables 56
B.2.3 Tree Construction 57

C SQLite Development **59**
C.1 Input script used for the experiment 59

D Potential New Opportunities **60**
D.1 Fine-grained Unit-testing of functions 60
D.2 Quorum-based problem-solving 60

E The Problem with Threads **62**
E.1 Nondeterminism Case-study 62
E.2 Contention 63

Chapter 1

Introduction

1.1 Overview

Computer execution performance has not progressed much and execution speed in terms of clock cycles have stagnated. The serial execution of instructions in a processor has not advanced in the recent past. *Figure 9* in *[Smith et al., 2012]* displays quite clearly that raw processor speed has plateaued.

The answer to the above problem is to approach problems non-serially, either using a parallel or concurrent approach utilising multiple computers. At one end of the spectrum there is *multithreading*, which is parallel computation in which all parallel executions take place in the same program within the same memory-space. Most *Operating Systems* (OSes) support execution of multiple threads or tasks at a time, with special support for multicore processors optimised for multiple threads of execution at a time.

The other end of the multiprocessing spectrum is the *message-passing* model, in which the separate executions share absolutely no memory with each other but synchronise the computation efforts using messages sent back and forth between nodes comprising the cluster.

Between these two multiprocessing models lie a number of approaches that are neither strictly *message-passing* nor *shared-memory* threaded. These other mechanisms are covered thoroughly in *Chapter (2)*).

1.2 Proposed Mechanism

I propose that a multiprocessing mechanism using neither multiple copies of the same program, *nor* being limited to a specific programming language, *nor* simply calling remote services would pose some interesting new avenues of research.

In brief, I propose that new opportunities may exist in the event that it becomes possible to write a single program that executes on multiple computers. To achieve this I propose creating a mechanism to transmit and execute a local native function in a running program to a remote node. This mechanism would function on such a foundational layer of the software stack that it would be considered a system-level service.

This would mean that such a mechanism would have to be provided as a system library that can be used from virtually any programming language on the system and any service offered by the system. Such a requirement precludes the usage of a non-systems programming language as the development language. In *Linux* such a library is written in either *C* or *Fortran* and the deliverable is a shared system library matching the filename pattern `lib*.so`.

The ability to execute local functions on a remote computer may lead to a single large program that runs on multiple computers rather than (as is the case now) multiple individual programs running on multiple computers. Conceptually, these are two very different approaches: a single program that selectively executes parts of itself on remote computers vs multiple programs that are executed on multiple computers.

1.3 Research Questions

A few of the more interesting questions raised by the existence of such a mechanism are:

- What mechanisms exist to support the distribution and invocation of native-execution functions?
- Is a *message-passing* design necessary for general purpose clustered and/or grid computing if *remote-function execution* exists?
- Are there measurable differences in presenting a grid of computing nodes as *a single computer to a single program* instead of as *a collection of computers to*

a collection of programs?

- What are the effects on software design and architecture when changing from a *message-passing* strategy to a *remote-function execution* strategy. Specifically and importantly, what new opportunities can we exploit using this mechanism?

The questions naturally lead to the following objectives:

Core Develop a mechanism supporting the ability to invoke and/or execute a remote function across multiple computers.

Core Benchmark the developed system against an existing industry method to gain an idea of the comparative performance of remote-function execution.

Advanced Use the remote-function system to run an existing and unmodified program in such a way that it uses a cluster.

Advanced Demonstrate the opportunities that arise when the ability to execute remote-functions is a reality.

This project, preliminarily named *Rexel* (***R**emote **Ex**ecution **E**nvironment for **L**inux*), is an altogether new approach to executing programs across multiple computers and aims to provide an easy and scalable mechanism to distribute snippets of a program across multiple computers. The snippets of code are pieces of native code encapsulated as individual functions callable from within the systems native libraries.

1.4 Process

This project did not involve human subjects in any way.

The flow of this thesis is fairly straightforward. Firstly, an examination of the current strategies for distributed computing is presented (*Chapter (2)*). The literature dealing with the current strategies is presented along with each multiprocessing method and a relevant existing example of the strategy.

Chapter (3) focuses on presenting a detailed technical view of the implementation of *Rexel*, including the rationale behind many of the design and engineering decisions. Some of the constraints that steered the direction of the development is also mentioned. The successful implementation of remote-function execution is a core objective of this thesis.

Thereafter another core objective, the results of a comparison between *Rexel* with *MPI*, is presented in *Chapter (4)* and an advanced objective is presented in *Chapter (5)*, along with a few thoughts on further possibilities that may be exploited by a system such as *Rexel*.

The final chapter, *Chapter (7)*, focus exclusively on the knowledge gained from this research and on the conclusions drawn from all the research carried out.

Chapter 2

Multiprocessing: Current State of the Art

There are many existing solutions that aim to distribute computational load across multiple computers. The most basic method is to spawn a number of processes each *reading* sections of the input data. Another method is to create multiple threads of execution, all of which share the same memory space. Orthogonal to the *shared-memory* approach is the *share-nothing[Mitrović et al., 2013]* approach in which none of the threads share any data; they all pass messages to each other instead.

In between these two extremes we have a number of different approaches to the problem of computational load-balancing across a cluster of nodes including hybrid systems which augment *message-passing* with a complicated, if limited, *shared-memory* model. *[Hoefler et al., 2013]* presents such a system while also providing informative coverage of other attempts at implementing *shared-memory* with *MPI*.

The most primitive distribution of a computing load is the distribution of storage across a cluster of nodes. In this scheme multiple nodes are consolidated to present a single large datastore. An example of this is the distributed directory implementation of *OrangeFS[Yang et al., 2011]*.

Another approach is the creation and submission of *batch jobs* to a master node which then schedules the jobs on remote nodes as specified by the operator. This often requires knowledge of the individual nodes in order to afford the operator the ability to optimise their submissions. For example, the operator may have to specify the minimum local storage or the minimum local memory needed for the batch job. An example of a system that uses a batch submission approach is the Globus*[Foster et al., 2008]* grid-computing platform.

At a higher level of abstraction are those mechanisms that offload general purpose programs onto remote nodes via system shell scripting or similar. These programs, such as *Gnu Parallel* and *Ansible[Hochstein, 2014]*, will take a single local program and execute it on a known remote node. They require little setup and can work with most programs written for a POSIX compliant platform. Due to the general nature of these programs they are limited in features and scalability.

Some applications are designed specifically to spawn invocations of themselves on multiple remote nodes. They have no need of the general purpose programs mentioned above and have the necessary mechanisms built in. These applications, such as *Distcc[Clemencic and Couturier, 2014]* and *Gnu Make* can spawn themselves to work in parallel when directed to do so*[Charwat et al., 2013]*.

Remote Procedure Calls (*RPC*) offer a slightly higher level of abstraction. A program written to use RPC system libraries can take advantage of services offered by a remote computer. They are the most common form of distributed computation due to being the foundation of many unix services*[Peter et al., 2014]*. The *Networked File System* (NFS) which uses RPC as a foundation forms the basis of many *High Performance Computing* (*HPC*) clusters.

More sophisticated mechanisms are those that offer both a framework as well as native libraries specifically for large-scale parallel computations. An example is the *Message Passing Interface* (*MPI*)*[Goodell et al., 2011]* that is currently the most popular form of parallel computing on *HPC* clusters. Just about any large scale computation uses an implementation of *MPI* that conforms to the standard*[mpi forum.org, 2015]*.

Instead of language enhancements, as in *MPI*, there are concurrent execution mechanisms that go even further and have native language support built into the language. One of the most popular languages for large-scale concurrency is Erlang*[Aronis et al., 2012]*, which is a language containing concurrency primitives such as message-passing.

Newer language designers have opted for an even more sophisticated mechanism by using a virtual software runtime as the target of the language rather than compiling the language down to native code. This extra layer allows the runtime environment to present itself as a single resource even when it is distributed across multiple nodes, thus allowing the programs written for the runtime to also be distributed in a transparent (to the developer) manner. The *Java Runtime Environment* is probably the most well-known language runtime that offers this feature. Infinispan and

Hazelcast*[Kathiravelu and Veiga, 2014]* serve as examples.

Even more specialised are those software products that are built specifically to support a certain class of concurrent algorithms. For example *Apache Hadoop[White, 2012]* is designed to support MapReduce*[Brown et al.]* solutions and fare poorly when used with solutions other than the intended one.

Then there are the specialised hardware solutions, which use the multiple processors available on discrete computer graphics accelerators for less specialised computation*[Mivule et al., 2014]*. These approaches are optimised for highly specialised tasks involving large numbers of floating-point computations running on specialised processors, either GPU, FPGA or ASICs*[Taylor, 2013]*.

Finally, the most complex and ambitious mechanism: Distributed Objects. Using a *Network Object Request Broker*, a caller can instantiate unique objects on remote nodes. While *CORBA* usage (and literature associated with usage) has fallen, Distributed Objects still live on in technologies such as XPCOM*[Onarlioglu et al., 2013]* and Java RMI.

The above is the current state-of-the-art in distributed processing. Each of the above technologies fall into one of the following categories:

- Multiple programs running on multiple computers work together towards an objective, or
- A single program can request services from other nodes within the cluster, or
- A virtual platform in the form of a runtime environment is used to execute that platforms code on remote nodes which are running the same virtual platform and runtime environment.

In summary, the current state-of-the-art in distributed computing works only with a dedicated runtime environment *OR* uses a standard to specify copying and executing whole programs on remote computers. None of the literature suggests mechanisms to push code snippets to remote nodes and then execute them.

This project focuses very narrowly on distributing native-code functions across multiple nodes. As such, it can be used to provide many of the functionalities in the products and research mentioned throughout this chapter.

The artefact produced is expected to be foundational, used to provide the ability to create new tools, and not necessarily as a solution to a specific existing problem.

Chapter 3

Core Objective: *Rexel* Implementation

3.1 Goals

The goals of the proposed technology artefact turned out to be critical requirements which influenced the choice of solution. Any solution that was unable to achieve any one of the goals listed below was discarded.

Take any local native-code function and execute that function on any number of remote nodes.

Allow any and all programs on the system access to this functionality, regardless of source language used to program the application.

Prevent any dependency on a particular language. All dependencies should extend no further than instruction-set and OS *combinations.*

The final goal listed above proved useful in exploring new possibilities; the experiment in *Section (5.1)* would not have even been conceived off had *Rexel* been dependent on a particular programming language.

3.2 Requirements

To properly answer the questions posed in *Section (1.3)* a technology artefact was planned for development. Based on the objectives of this thesis and the research questions in *Section (1.3)*, the following minimal requirements for the proposed system, *Rexel*, were:

- Allow execution of *native* code functions on a remote node,
- Allow the determination of the distribution strategy to be made at runtime,
- Provide system-level access to this mechanism, i.e. access to this mechanism should not be limited to a particular programming language.

The above requirements ensure that any technical artefact produced will fulfill the goal of running a single program across multiple computers, leading to meeting the stated objectives and resulting in answers to the research questions.

Additionally, no requirement was made to allow cross-platform portability of the resulting code-snippets - this is impossible as the code-snippets were to be native-code, for a specific *OS* and hardware combination.

As a further relaxation of the constraints, no requirement was made for encrypted or secured communications. Secure communications can be added to any communications channel using a secure tunnelling protocol, hence security is out of the scope of this implementation.

3.3 Overview of Concept

The chosen method of implementing remote-function execution is, broadly speaking, a simple enough concept: transmit the specified library file containing the function over to the remote computer, then instruct the remote computer to selectively load and execute functions within this library. Results from such an execution can then be transmitted to the calling computer.

While this concept is simple enough to grasp in a couple of jargon-free sentences, there are immediately obvious issues in performing the remote-function invocation. Leaving aside the obvious security issues, the main question is, even if there exists a program on the remote computer that can store and then load the specified library file, how can the program know what parameters any given function takes?

Other issues that appear daunting are those arising from the fact that there are restrictions, even in individual programs, inherent in calling functions that are not available for compiler examination during the building of the program. Most issues were overcome, and a few issues were worked around in an acceptable manner.

3.4 Architecture and Execution

The *Rexel* framework is split into two separate and distinct programs: a `rexel-node` program which runs on the remote computers and serves *Rexel* requests, and a `librexel-master.so` library which client programs must link with at runtime in order to make use of the *Rexel* framework.

The `rexel-node` program is a simple fork-based socket server that listens on specified TCP ports for incoming instructions. Once an instruction is received the corresponding function within `rexel-node` is executed using the parameters that followed the instruction.

The communications protocol is a simple one and comprises merely a command and list of parameters for that command. Each of the elements in the request is packaged as an `atom_t` structure that is capable of representing all primitive *C* datatypes. The number of parameters present in a request is dependent on the command used; for example the command to get status (percentage of RAM used, total processor load, etc) takes no parameters, while the command to execute a particular function takes virtually unlimited parameters. A comprehensive description of the protocol used is present in the header files `protocol.h` and `atom.h`.

The *master* program is any program written to use the `librexel-master.so` library. This library contains all the functions necessary for a program to connect to, and use, `rexel-node` programs on remote computers. The `librexel-master.so` library contains a constructor function that reads in configuration files on library load (see *Appendix A*).

Once the constructor has loaded all the known nodes, the client program will start executing from `main()` and all the *Rexel* functionality will be available. Note that the client need not do anything other than perform dynamic linkage with *Rexel*. The constructor ensures that before the client program even starts executing, all of the *Rexel*) configuration is complete.

In the intended usage, the client program will call a *Rexel* initialisation function that specifies which namespace to use. Only nodes from the given namespace will be used. The client can then call functions such as `store_library()` which will take a given local library and transmit it to all the remote nodes in the namespace for storage, or `register_function()` which will inform all the remote nodes in the namespace to register the fact that the specified function exists in the specified library. After this is done the client can call functions that will execute the specified library on the remote nodes using the given parameters. Both synchronous and asynchronous variations of `callfunc()` exist, with timeouts that can be specified so that if a remote node crashes the local function will not wait forever for a response.

On the remote computer's end, the `rexel-node` program will store any library sent to it and call any function in a given library (if that library was sent to it - local libraries on the remote computer are ignored by `rexel-node`). When a request is received to register a function, `rexel-node` both loads the library containing that function into memory and generates a template for future function calls using the prototype that is sent with the function registration request.

The prototype is generated as a template suitable for use with `libffi`*[Lang, Green, Löscher and Sagonas, 2016]*, a portable library that enables the caller to call arbitrary compiled native code functions at runtime. When a request to execute a function is received, `rexel-node` populates the parameters to `ffi_call()` with the parameters from the request, executes `ffi_call()` and decodes the result using the prototype that was specified in the function-registration request. The result is then sent back to the originator of the request over the open TCP port.

3.5 Challenges

One of the first challenges in producing the technology artefact that became the *Rexel* framework was the issue of calling arbitrary compiled native-code functions at runtime. The initial solution consisted of using the system's dynamic loader and linker functions, `dlopen()` and `dlsym()`. This solved most of the problem, in that it allowed `rexel-node` to locate a specific function in a specific library at runtime. If the function was compiled for a different platform then the `dlopen` function would fail with appropriate error responses. If the function did not exist (or there was a problem with the location of the function within the library), then `dlsym()` would fail with appropriate error responses as well.

Having located and loaded the specified function, there remained the issue of calling it. Forcing a function-registration step prior to a function call allowed `rexel-node` to ensure that it had a prototype for the function describing the primitive types of each parameter to the function and the type of the return value of the function. With this information, all that remained was to call the correct `libffi` functions specifying the parameter types and the return type.

However, there were still some restrictions. For example, `libffi` did not support the calling of functions that took a variable argument list. The reasons for `libffi` not supporting these are simple - the *C* standard does not support all primitive *C* types in variable argument lists. For example, the floating-point type `float` is passed into a function not via the stack but via floating point registers. When the instructions for a function call is generated, the compiler will know from the function declaration whether it is a variadic function or not. If variadic, then the `float` type is promoted to `double` and the function being called **must** use the value as a `double` and not a `float`.

A similar situation exists for `char`, `int8_t`, `short`, `int16_t` and all of their `unsigned` equivalents. These types are all promoted to `int` or `unsigned int` depending on their sign. The reason for this being part of the *C* standard is due to legacy compilers that promoted any integer literal to an integer if the literal would fit in an integer. They did this because the compiler, at the point of generating the instructions to call an undeclared function, would not be able to tell that the function `foo` took an integer when the function call was `foo(5)`. Promoting anything smaller than an integer to an actual integer allowed all these functions to work.

This presented somewhat of a problem, in that `rexel-node` supported a developed type called `atom_t` that could accurately represent literals smaller than integers, and accurately represent `float`s and `double`s distinctly. In the end, the decision was made to conform to the *C* standard's method of managing function call type promotions rather than attempting to fix them (and introduce complexity where none is really needed).

Another challenge that appeared roughly midway through the project was within the `rexel-node` architecture. Initially the `rexel-node` program was designed as a multithreaded server, in which incoming connections were handled by a thread spawned for just that connection. Due to performance-issue necessity all sockets (for all threads) were handled with a single `poll()` system call. This meant that each worker thread had to communicate to the master thread to receive incoming data and transmit outgoing data, and involved the use of mutexes to lock certain data structures when data was incoming or outgoing. Under high loads the locking severely degraded the performance of the `rexel-node` program to the point that unrelated threads were losing network data.

The solution was to redesign `rexel-node` to use a `fork()` based architecture in which each incoming connection was handled by a `fork()`ed process. This kept performance within reason as the network data then was only copied once, from kernel to process, and not from kernel to master-thread, and then to child-thread. The lessons from *[Lee, 2006]* were thoroughly re-learned in this exercise: the added complexity of mitigating each of the problems presented by using threads in a non-trival fashion almost derailed the project before it was properly started.

Threads are still used in parts of *Rexel* to achieve concurrency when talking to multiple remote computers, albeit in a very limited and thoroughly inspected and tested manner.

On a positive note, the strict testing that accompanied the development of each module significantly helped smooth the development effort. *Rexel* and all the associated programs consists of around 14000 lines of code containing some 350 functions written in *C*. A test-harness was used to perform automated regression testing after each build, along with an automated method of checking that the tests passed using the Unix `diff` program to determine differences between test executions. The unit-test routines were all written to verbosely describe the action and result for each test of each of the ≈ 300 functions that were tested.

3.6 Summary

The *Rexel* framework, as an artefact, meets all the stated goals. The artefact developed has performed as envisioned and, importantly, allowed further investigation into the more interesting academic problems mentioned in the objectives.

All of the subsequent thesis objectives were dependent on a complete and successful implementation of remote-function execution; *Rexel* was used as the implementation.

Rexel*, as it currently stands, represents a complete achievement of the objective: develop a mechanism supporting native-code remote-function execution.*

Chapter 4

Core Objective: Comparison of *Rexel* and *MPI*

4.1 Introduction

MPI is the de-facto standard in *HPC* applications*[Cardellini et al., 2016]*. It has had over two decades of constant refinements*[Cardellini et al., 2016]* and improvements and is usually regarded as the first option for high-performance multi-computer applications.

It is for these reasons that *MPI* is chosen as the frame of reference when benchmarking *Rexel* performance. To be clear, no expectation of improving upon *MPI* performance was held when designing *Rexel*. *MPI* performance is used only as a reference upon which to judge whether *Rexel* is performing adequately and/or similarly to *MPI*.

The problem chosen is a search problem with sparse results in a large search-space, namely finding prime numbers up to a certain limit using trial division. Implementations for both *Rexel* and *MPI* are written based on a similar, almost identical, implementation of trial division.

4.2 Executable Programs

Insofar as it was possible, both programs use the same algorithm with the same compiler and compiler flags being used to generate the executable programs. Both programs were compiled on the same computer at the same general time and both programs target the same hardware and *OS* combination.

No advantage was knowingly given to either implementation of the trial division algorithm. Both programs use the trial division approach to find prime numbers as displayed in the following pseudocode:

```
1. limit = <limit determined by user>
2. factor = 2
3. start loop
4. count primes from 1..factor
5. record duration while counting
6. factor = factor × 2
7. if factor >= limit then end
8. goto {3}
```

Note that the above algorithm uses two limits for the search-space. The first is the outer limit and is determined by the user. The second is an internal limit and is preset within the algorithm. The program simply begins searching for prime numbers up to the internal limit, and then repeats the search with the internal limit doubled. The program ends when the internal limit exceeds or meets the user-determined external limit.

The reason for this is because time on the cluster is limited and it was desirable to have an easy way to change the amount time a single execution took in order to fulfill the intention of having three test-runs.

The *Rexel* implementation of the trial division algorithm is listed in *Code Listing 4.1*; the *MPI* version of the same algorithm is listed in *Code Listing 4.2*. The differences between the two functions are due to the *MPI* version needing to be *MPI*-aware in order to determine the portion of the search-space to work on. The *Rexel* version does not need to be aware of the environment it is running in, although it could be written to be *Rexel*-aware if performance gains were a priority. In *Rexel*, the code executing on each node may use the context as an option but it is not necessary.

This algorithm was chosen due to the performance profile of $O(n^2)$. An exponentially increasing duration was desirable as it allowed the search-space to be increased by a factor of two multiple times during a single invocation of the program. Absolutely no optimisation of the algorithm is performed.

Listing 4.1: *Rexel* Trial Division algorithm

```c
uint32_t internal_prime_count (uint32_t from, uint32_t to)
{
   uint32_t total = 0;

   for (size_t i=from; i <= to; i++) {
      total++;
      for (size_t j=2; j < i; j++) {
         if (!(i%j)) {
            total--;
            break;
         }
      }
   }
   return total;
}
```

Listing 4.2: *MPI* Trial Division algorithm

```c

int prime_number ( int n, int id, int p )
/************************************************************************
  Parameters:

    Input, int N, the maximum number to check.

    Input, int ID, the ID of this process,
    between 0 and P-1.

    Input, int P, the number of processes.

    Output, int PRIME_NUMBER, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total;

  total = 0;

  for ( i = 2 + id; i <= n; i = i + p )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( ( i % j ) == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

4.3 Methodology

Care was taken to ensure that the results were all accurately captured and no results were overwritten or lost.

The prime-counting programs both work identically and, save for the informational messages printed to screen, produce identical output in the form of three columns of numbers containing:

1. The limit of the search-space,
2. The number of prime numbers found in that search-space, and
3. The duration of time it took to exhaust the search-space.

An example of the output is given in *Code Listing 4.3*. The format of the output enabled easier data aggregation during the analysis of the data.

The testing consisted of three test-runs for each of the programs, with the upper limit of the search-space set to 524588. A fourth test-run using an upper limit of 1048576 was performed after the first three test-runs as all three test-runs executed faster than expected and thus there was still time left to use the cluster.

Each test-run consisted of executing each program on the cluster 10 times, starting with a cluster-size of one node and ending with a cluster-size of 10 nodes. All of the execution was automated (the shell scripts responsible for starting each program is available in the git repo) and output from every execution was captured in a text file named with the method (*Rexel* vs *MPI*), the limit specified and the number of nodes used.

In short, for each test-run, each program was first run with {limit=524288, nodes=1}, then {limit=524288, nodes=2}, etc until the program executed {limit=524288, nodes=10}. A final test-run using {limit=1048576, nodes=[1..10]} was executed for each program.

4.4 Test Setup and Architecture

Each test-run was run on the same hardware and software combinations. To produce a cluster a single *VM* (Virtual Machine) was created on a borrowed *VMWare* server that was designed and built specifically to host isolated *VM*s. This setup is representative of most cloud-providers' Virtual Private Server offerings.

Listing 4.3: prime_mpi output

```

PRIME_REXEL
A rexel program to count the number of primes.
Running on 2 nodes
         2         1       0.003288
         4         2       0.006597
         8         4       0.005358
        16         6       0.003974
        32        11       0.004020
        64        18       0.007877
       128        31       0.004159
       256        54       0.007831
       512        97       0.004252
      1024       172       0.004157
      2048       309       0.004442
      4096       564       0.005991
      8192      1028       0.014740
     16384      1900       0.039568
     32768      3512       0.138739
     65536      6542       2.491621
    131072     12251       3.092161
    262144     23000       7.175698
    524288     43390      27.195284
```

A total of 11 *VM*s were created: the first was used as the master and the other ten were used as nodes in a cluster. To create the nodes, a single node was first created with all the required software (including the benchmarking software used in this test). This single node was created with 128MB of RAM and 1.2GB of hard-disk space. It was installed with Linux Debian 8.0 (Jessie) and all unnecessary software was removed (print servers, GUI support, etc).

After the first node was tested to ensure full and correct functioning it was shutdown and then cloned another nine times. This ensured that every single node was identical to every other node.

To reduce the possibility of network traffic from outside the test interfering with the test, all of the ten cluster nodes were created within their own virtual network that exists only inside the *VMWare* server. In other words, the actual network packets never traversed a network and thus network speed and latencies could not affect the execution of either *Rexel* software nor *MPI* software.

Further, the single master node in addition to being connected to the general network was also logically connected to this virtual cluster-only network via a separate network interface, also on the same *VMWare* server. This meant that communications between the nodes and each other and the nodes and the master were never affected by actual network traffic. This is illustrated logically in *Figure 4.1*

All interaction with the nodes were from the master only as the nodes had no

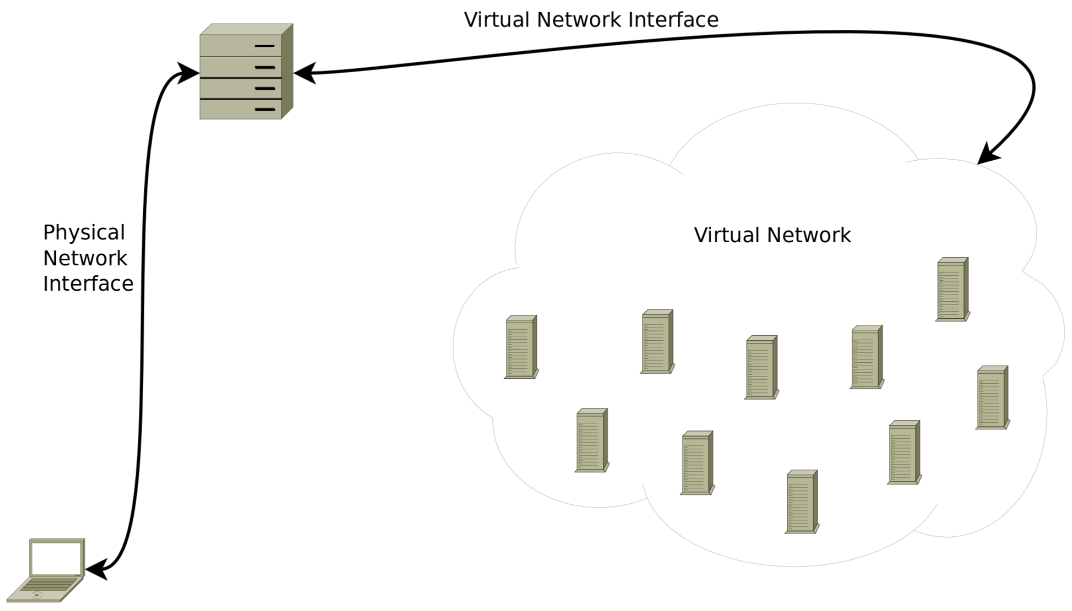

Figure 4.1: *Network Diagram showing layout of the network for testing.*

access to a real network. All interaction with the master were via an *SSH* session initiated on the master's general network interface so that no communication with the master while tests were running would interfere with the nodes' communications.

The master did not run any computation itself and this was specifically to ensure that no latencies or delays were introduced in capturing the output of each node. For the *Rexel* test the main program ran on the master while executing test functions on the nodes. None of the test functions were executed on the master itself, even though this is possible with the *Rexel* architecture, because of the desire to ensure that the master correctly and timeously captured each nodes output.

The *VMWare* server was used at a time when load on the computing infrastructure was expected to be very low (Friday afternoon after all employees had left for home).

4.5 Capturing and Preparation of Data

The results of each test-run were captured in a file representing the method used, the number of nodes used and the limit specified. As an example the results for executing the *MPI* program on four nodes with a limit of 524288 were captured in a file called `mpi-run.4.524288`. The files resulting from each test-run are stored in their own directory named from `1` to `4`. Hence, the results of test-run `2` using the *MPI* method on four nodes with a limit of 524288 can be found in the git repo under the name `2/mpi-run.4.524288`.

To amalgamate the data into a single data source a shell script `results2csv.sh` was written to read all the data files and produce a *CSV* (Comma Separated Value) file that most spreadsheet software can import. The shell-script `results2csv.sh` as well as the output of the script, `results-all.csv` is available in the git repo as well. The file `results-all.csv` was then used as the master data source for all the other analysis and operations.

Firstly, the file `results-all.csv` was imported in a spreadsheet program and the spreadsheet program was used to generate some general statistics. The spreadsheet file was then saved and is available in the git repo within the same directory as all the other results.

Thereafter data for each of the *internal* limits (see *Section (4.2)*) was pulled from the master data source using a shell-script specifically written for this task, `genmat.sh`. This shell-script identified each internal limit within the master data source file for each unique combination of method and internal limit. These records were then passed to an external program which was written to accept these records and return the mean and standard deviation for each record specified. The external program is in the git repo as well, under the name `statistics.lisp`. The results of all the executions of `genmat.sh` were written to a secondary data source file called `stats-per-searchspace`.

Next, the statistics for each method were captured from the file "stats-per-searchspace" into two new files, `mpi-stats` and `rexel-stats`. These two files, being *CSV* files, were then imported into the spreadsheet where they were placed into rows and columns in a manner suitable for the spreadsheet's charting capabilities.

After an initial analysis of the data (including the generation of the charts), two more shorter analysis took place due to unexpected outliers in the data. This is discussed in more detail below in *Section (4.7)*.

Finally, the spreadsheet was saved again in the git repo and was used to generate the charts of all the results. All of the final data files, including the spreadsheet, are in the git repo along with the master data source file, the raw data generated by the programs and all the tools necessary to regenerate all the intermediate data files and final data files which can be used to repeat the analysis at a later stage.

4.6 Analysis of Results

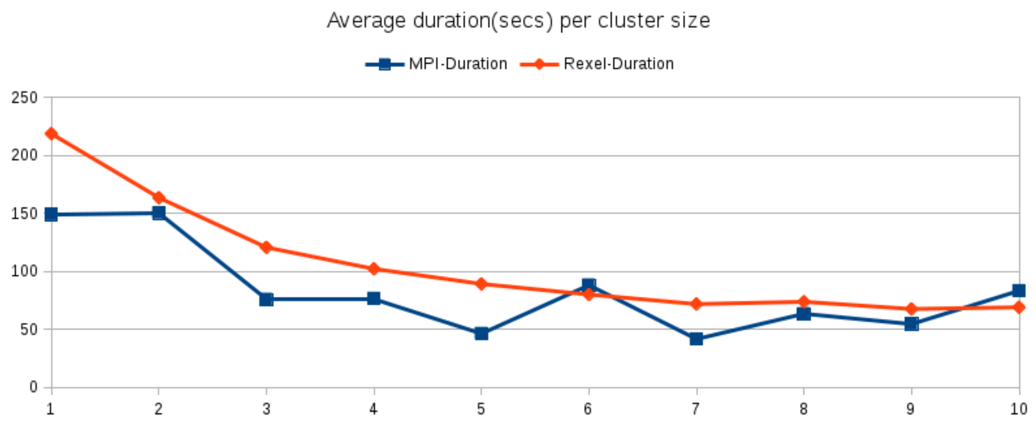

Figure 4.2: *Change in duration as cluster-size increases.*

As detailed above, each test-run consisted of a number of individual invocations of the program with cluster size varying from a single node cluster to a 10-node cluster. This allows us to compare the performance differences as the cluster size changes for both *Rexel* and *MPI* methods.

The results of all four test-runs for all user-defined limits were averaged to produce the chart in *Figure 4.2*. As expected for both the methods tested, the addition of nodes to the cluster correlated strongly to a performance increase. While the *MPI* method more often than not outperformed the *Rexel* method the *Rexel* method appeared to be more consistent and predictable in optimally using additional nodes as seen in *Figure 4.2*.

The results are not fully unexpected; after all it is expected that the performance will increase as nodes are added to the system.

The number of nodes per cluster was not the only variable that was tested. The other variable being tested was varying loads, implemented by means of increasing the search-space for prime numbers. The algorithm (see *Section (4.2)*) tested up to limits in powers of two, i.e. the algorithm first tested a search-space up to 1, then 2, then 4, then 8, then 16, etc until it reached the user-defined limit. The user-defined limit is 524288 for the first three test-runs and 1048576 for the final test-run.

The results for each of these search-spaces were grouped by search-space and statistics were generated for these groupings as explained in *Section (4.5)*. Due to the linear scale of the charts the search-space durations and deviations for the smallest and the largest search-space would not comfortably fit on a single-scaled chart. To this end, the charts have been divided into two to display the results of varying the search-space: search-spaces up 16384 and search-spaces larger than 16384.

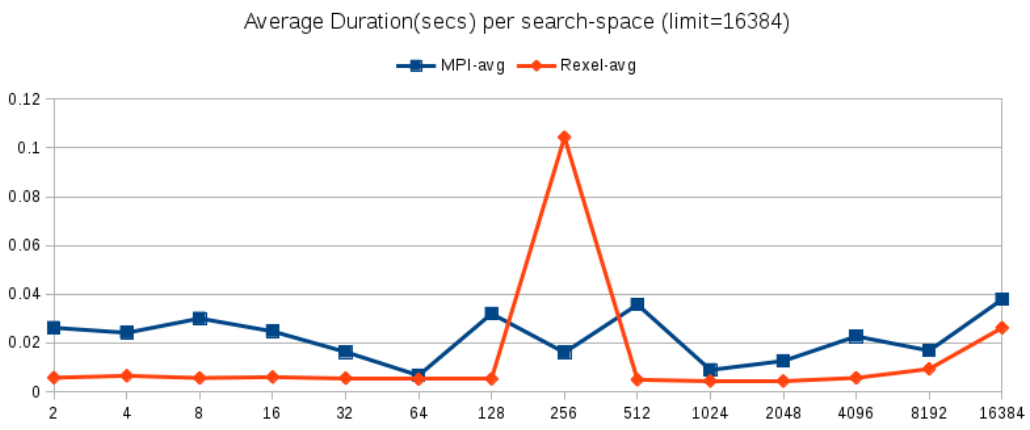

Figure 4.3: *Change in duration as search-space increases (up to 16384).*

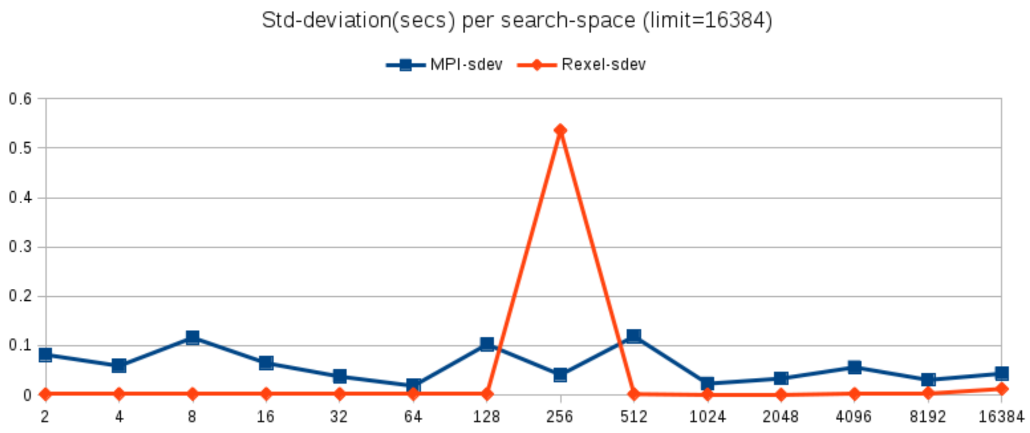

Figure 4.4: *Change in deviation as search-space increases (up to 16384).*

The results in *Figure 4.3* and *Figure 4.4* show that, with a single exception at search-space of 256, the *Rexel* method consistently outperformed the *MPI* method in terms of both duration and consistency at search-spaces below 16384. However *MPI* caught up in terms of performance and consistency at search-spaces of 32768 and surpassed *Rexel* above search-spaces of 32768 as shown in *Figure 4.5* and *Figure 4.6*.

As *Table 4.1* shows, *Rexel* performed well below *MPI* for the interval from 32768 to 262144. The performance of *Rexel* at search spaces above 262144 was still acceptable as the *Rexel* results were well within 20% of the *MPI* results, and was approaching the *MPI* results as seen in the final three results sets for search spaces above 262144 (displayed in *Table 4.1*).

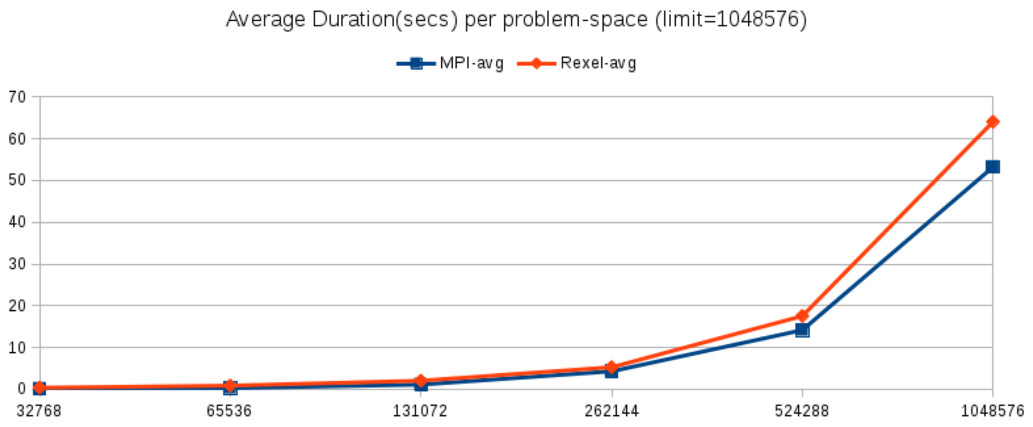

Figure 4.5: *Change in duration as search-space increases (up to 1048576).*

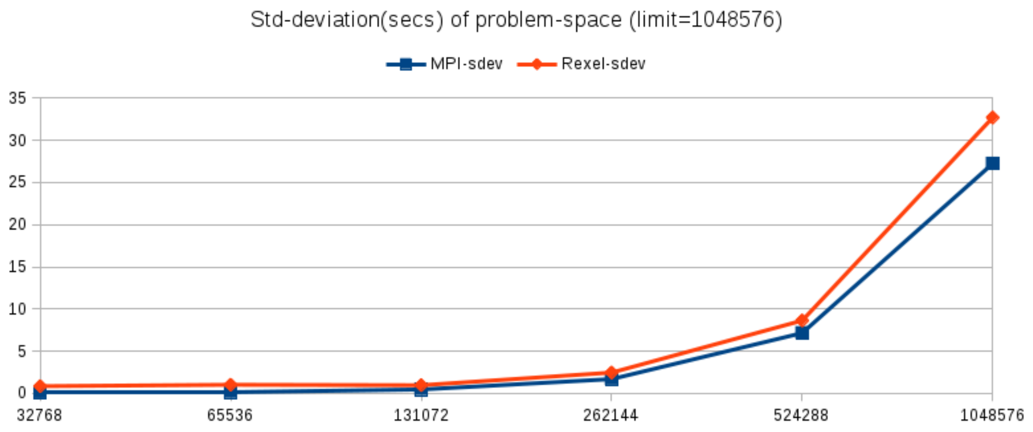

Figure 4.6: *Change in deviation as search-space increases (up to 1048576).*

4.7 Outlier Discovery

It was discovered during the analysis and generation of statistics that one of the test-runs during *Rexel* execution had unusually high average duration. Upon looking into the data it was found that during the execution of test-run number three across a nine-node cluster, a single node had apparently encountered some sort of problem and had taken over 3 seconds to respond (over 750 times longer than its peers).

This can be seen in *Figure 4.3* and in *Figure 4.4* as a large spike at the search-space limit of 256. Unfortunately the problem that lead to this spike could not be diagnosed as this anomaly was only discovered days after the cluster had been shut down.

Further complicating the issue is that that the extreme magnitude of this outlier was significantly skewing the results for *Rexel*. To gain a clearer view of *Rexel* performance another set of statistics per problem space was generated that excluded test

Limit	MPI-avg	Rexel-avg	% Diff
32768	0.1211203	0.3725165	67.486
65536	0.3202916	0.8717096	63.257
131072	1.1217341	2.0540545	45.389
262144	4.2990723	5.2816940	18.604
524288	14.173408	17.534016	19.166
1048576	53.249565	64.070700	16.889

Table 4.1: The margin between *Rexel* and *MPI* performance at the higher end of the search-space.

number three completely. This new set of results were the aggregation of tests one, two and four only. Comments were left in the script `genmat.sh` with instructions on how to go about regenerating this result-set.

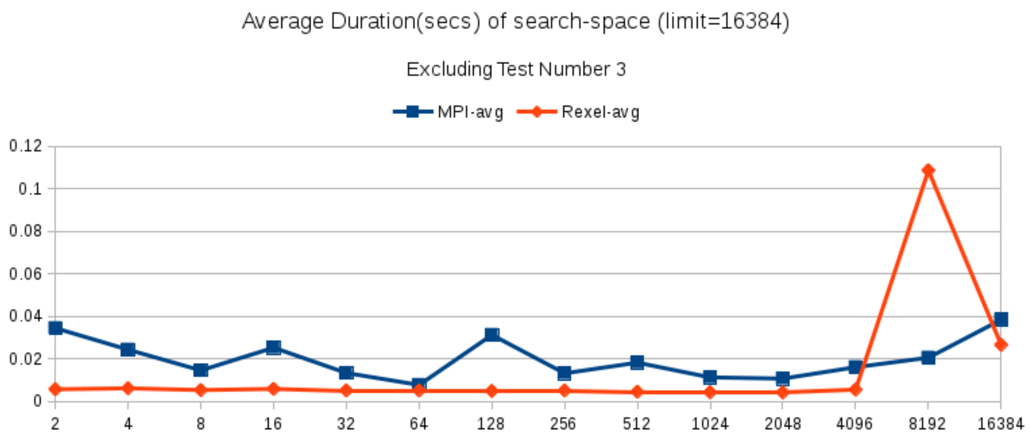

Figure 4.7: *Change in duration as search-space increases, excluding test three (up to 16384).*

The revised statistics excluding the outlier are presented in *Figure 4.7* and *Figure 4.8*. As can be seen in *Figure 4.7*, *Rexel* was performing very satisfactorily compared *MPI* with the outlier removed. A further outlier was discovered in this revised dataset at a search-space limit of 8192.

As this outlier had little effect on the ability to visually inspect *Figure 4.7* it was decided that no further cropping of outlying test results need be done. The problem was that this new outlier presented the viewer of *Figure 4.8* problems in visually inspecting the chart; it is impossible to see the difference between *Rexel*'s standard deviations compared with *MPI*'s standard deviations. As an aesthetic measure, purely to improve the visual representation of the standard deviations in this dataset, the outlying standard deviation in *Figure 4.8* was replaced with a linearly interpolated value (i.e. a value that lies midway between the previous data point and the next data point).

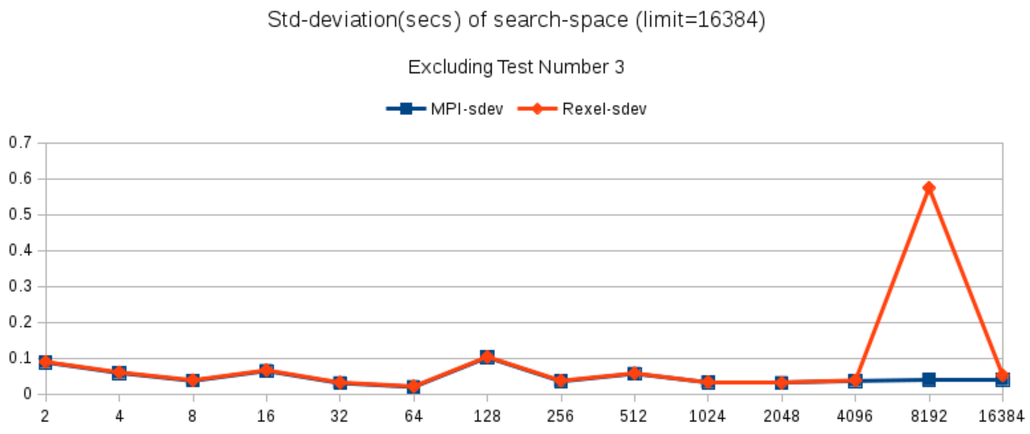

Figure 4.8: *Change in deviation as search-space increase, excluding test three (up to 16384).*

The result was a chart that was much clearer on relative consistency of *Rexel* and *MPI* as can be seen in *Figure 4.9*. To be clear, there is no attempt at data manipulation: the only reason to interpolate that single datapoint was to present a chart in which the differences in the standard deviations between *Rexel* and *MPI* were clear. The chart with the non-interpolated data point is presented in *Figure 4.8*.

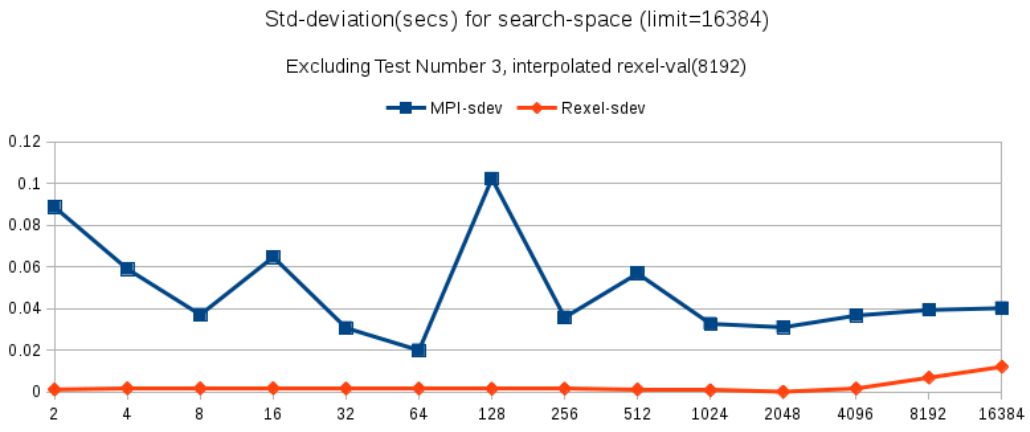

Figure 4.9: *Change in deviation as search-space increases excluding test three, interpolated @ 8192 (up to 16384).*

4.8 Summary

Rexel and *MPI* were tested together to determine *Rexel*'s comparative performance. The test setup consisted of a single master and ten nodes.

The nodes and the master were provisioned on a *VMWare* server that is representative of most cloud and *VPS* providers' setups. Communications between the nodes and each other and the nodes and the master were over an isolated virtual *VMWare* network. The master communicated with entities outside the cluster on a separate physical interface. All of the nodes were identical in hardware and software and the master did not run any computations and did not contribute to the benchmark.

Both methods employed in the tests solved the same search problem (finding prime numbers in a limited search-space) using a trial division algorithm. The algorithm was identical between the two methods employed with the exception of the *MPI* algorithm needing to be *MPI*-aware. The *Rexel* algorithm, designed as a local function that was in practice executed remotely, needed no such awareness of cluster-context.

A total of four test-runs were performed with the fourth and final test-run performed on an expanded search-space. Great care was taken during the test-runs to ensure that the ≈ 950 data points were all collected accurately. Data was collected for each search-space within each program invocation which was executed once only for varying cluster sizes between one and ten nodes per cluster. This was repeated for each of the four test-runs performed.

The results of the test-runs were aggregated to determine the average performance for each of the methods employed on different cluster-sizes. The results were further analysed to determine the average performance for each of the methods on different search-spaces. Further analysis to mitigate the effect of outliers were performed.

The analysis resulted in a number of charts visually depicting the relative performances of *Rexel* and *MPI*. A single data point was interpolated to reproduce one of the charts on a scale that allowed a more accurate visual representation of the remaining data points.

The conclusion drawn from the charts is that the *Rexel* implementation of the brute-force search solution offered a little performance advantage over the *MPI* implementation of the same, particularly in the larger cluster sizes and smaller search-spaces. The *MPI* implementation offered superior performance over *Rexel* in large search-spaces but negligible performance benefits in small cluster sizes.

Overall, for both large and small search-spaces and large and small cluster sizes *Rexel* performed with a smaller deviation than *MPI*. *Rexel* was consistent and predictable, both during the times it had the performance advantage *and* during the times it performed poorly compared to *MPI*. At all times *Rexel* was consistent.

The ends of the scales used, namely cluster sizes of eight, nine and ten and search-spaces of 262144, 524288, 1048576, turned out to be very interesting. At cluster sizes of eight, nine and ten the *MPI* implementation gained much more consistency, and at the search-spaces of 262144, 524288, 1048576 *Rexel* approached the performance of *MPI*.

The implication of the results of the upper ends of cluster size and search-spaces is that a larger test incorporating a larger cluster size and larger search problem could very well reveal a different performance and consistency profile to the results of the current test.

As a final point, it is worth noting that *MPI* and *Rexel* both have slightly different goals even as both aim to provide parallel and concurrent computational methods. *Rexel* aims to provide the ability for a single program to execute across multiple computers while *MPI* aims to provide a framework enabling the execution of multiple programs on multiple computers to solve a single problem.

In view of the slightly different goals the results of the comparative testing can be seen as something of a success for *Rexel*, which was written in a month by a single developer, as *Rexel* managed to mostly keep up with and in some cases even exceed the performance and consistency of the industry-standard *MPI* which has been refined and optimised by large groups of skilled experts over 20 years.

More optimisation efforts towards Rexel *might enable it to be more performance-competitive with* MPI *within the current test parameters.*

Chapter 5

Advanced Objective: Porting Unmodified Programs

5.1 Overview of Objective

This chapter deals with the objective of running an unmodified program, designed to run on a single computer, on a cluster of *Rexel* nodes. The reason for attempting such an endeavour with *Rexel* was to demonstrate a new technique that becomes available in the face of an ability to execute native-code functions remotely.

A secondary reason is to demonstrate the power and flexibility that is inherent in providing a system-level framework over a framework bound to a particular programming language (see *Section (3.1)*).

The goal of this experiment is simple:

Demonstrate remote-execution on an unmodified existing binary program.

The subject chosen for this little experiment is *SQLite[Owens and Allen, 2010]*, an embedded and self-contained *SQL* capable database. *SQLite* is a single-user, single-process, non-multithreaded-capable database which stores the entire database in a single file for convenience. It is extremely popular as an addition to standalone programs which need to store data but have no need for a full *Relational Database Management System* (*RDBMS*).

In addition to being used by standalone programs, *SQLite* comes with an interactive program which allows the user to query the database file (whether that file was created by the *SQLite* library or the the standalone program is of no significance).

5.2 Implementation Details

The program `sqlite3` was executed using input that created and populated a single table, then selected the entire table (The test script is in *Appendix C.1*).

The `sqlite3` program was then run with the table-creation input a second time while being monitored by the system program `strace`*[Stevanovic, 2014]*. The program `strace` monitors running processes and produces a log of all the system calls the process made along with the arguments passed to the system calls.

The output of `strace` was then searched for all the filesystem calls. The search revealed that `sqlite3` uses system calls `open()`, `close()`, `read()`, `write()`, `fstat()` and `lseek()`. New versions of some of the above functions were then written and packaged into a library, `libnetdup.so`.

The environment variable `LD_PRELOAD` was set to the value “`libnetdup.so`”. Setting the environment variable `LD_PRELOAD` in *Linux* causes the specified library to be loaded ahead of any system library, thus any functions that are found in the preloaded library are used in place of any functions of the same name from a system library. In this way, it is possible to replace specific system calls with a user-specified function.

The replacement functions written were `open()`, `close()` and `fseek()`. These replacement functions all used *Rexel* to perform the file operations they replaced both locally (by calling the local functions `open()`, `close()`, etc) *and* calling the same function on a number of remote *Rexel* nodes.

The intention was to intercept any file-modification calls made by the program, in this case `sqlite3`, and then call the same function remotely on a different computer. In this way, when `sqlite3` called the system function `open()`, it was intercepted by the replacement in `libnetdup.so` which then called the real `open()` system call, in addition to calling the `open()` system call on remote *Rexel* computing nodes.

In this manner, any file created or modified by `sqlite3` would be instantly and constantly replicated across the entire cluster.

The logical architecture of the system is shown in *Figure 5.1*, with the system call being made displayed in *Red* and the return value displayed in *Green*.

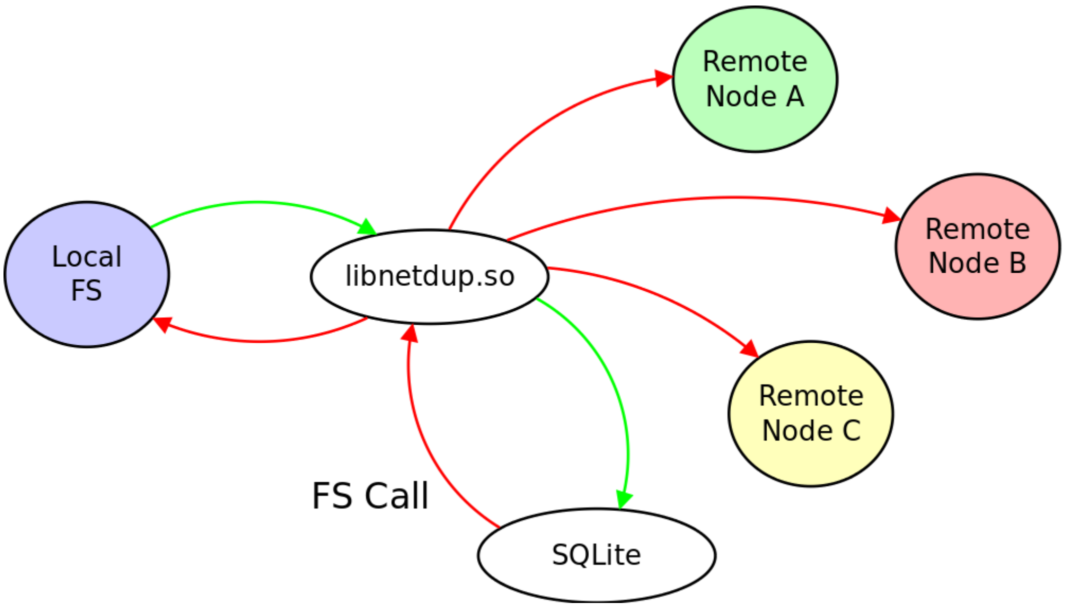

Figure 5.1: *Logical Model of the environment used to run* SQLite *on multiple nodes.*

5.3 Result of Experiment

This experiment produced a mixed result; on the one hand the unmodified binary program successfully utilised multiple nodes in a cluster (the actual test used four nodes) without being aware that it was running in a cluster. After the files on all nodes in the cluster were determined to be identical, this experimental development was considered a success.

On the other hand, the program `sqlite3`, as it stands (unmodified), is not aware of any remote computers or parallel function calls. All it knows is that it wrote (or read, or seeked) a file, and it expects a return code for that write/read/seek. The replacement functions do indeed return the correct results for the local filesystem operations, but not for operations executed on remote nodes.

The reason for not returning any values from the remote-function invocations for the filesystem operations is simple: there is nothing that can be done if the remote call fails. There is no way to return an error code to the `sqlite3` program that says, in effect, *"A remote function failed"*, because `sqlite3` is not modified to understand the concept of parallel filesystem calls, or of multiple processes.

Thus, because the program was not aware of multiple executions of the filesystem function calls, there was no way to inform the program about failure of any of the nodes. A little more thought was put into this, and to mitigate this in future executions of this nature the replacement functions in `libnetdup.so` can report the error to another place, perhaps the system log. This allows the administrator the ability to at least determine that a failure has occurred and take appropriate action.

A final note: although only filesystem function calls were intercepted in this experiment, *Rexel* is not limited to filesystem calls. Any library function, including ones that ship with a program, can be replaced in this manner so that certain functions are run in parallel on multiple nodes in a cluster. This functionality does not need any modifications to the binary program as was demonstrated using *SQLite*.

Large programs with computationally expensive functions would benefit from a similar effort to the one performed in this experiment for *SQLite*.

An unmodified binary program was successfully distributed across a cluster using remote-function execution.

Chapter 6

Advanced Objective: Partial Shared-memory Processing

This chapter discusses an arithmetic expression evaluator designed to offload subexpressions onto remote computers. This is performed using a *partial shared-memory* approach, differing from the traditional parallel computation architectures which use either a shared-memory approach or a message-passing system.

The goal is to marry the simplicity of use associated with the *shared-memory* approach *[Mivule et al., 2014]* with the scalability and robustness of the *shared-nothing* approach *[Mitrović et al., 2013]*. There are many problems with *shared-memory* systems using threads or similar*[Lee, 2006, Ousterhout, 1996, Parekh et al., 2000, Rinard, 2001, Sutter and Larus, 2005]*, not least of which is the fact that some of these problems are, in practice, unsolvable due to the non-determinism introduced by threads*[Lee, 2006, p.5]*.

There are also existing implementations of *shared-memory* and *message-passing* hybrids*[Hoefler et al., 2013]*. These systems implement full two-way *shared-memory* over *MPI* and therefore possess all the existing *shared-memory* problems, such as deadlocks and data-races, and introduce new problems, such as synchronisation*[Hoefler et al., 2013, p.8]*.

The implementation of this concept, code-named *The Uncommon Lisp Environment*, is comprehensively detailed in *Appendix B*. The source is available in the source code repository.

While there is no *shared-memory* nor explicit *message-passing* in the implementation discussed in this chapter, *implicit* use of *Rexel*'s remote-function execution facilities is made. On a physical layer there are indeed bits and bytes communicated between remote computers, so in effect there is a form of *message-passing* occurring, but the input to the interpreter does not reference *message-passing* in any way. The author of the expression will not have any indication that the expression is evaluated in parallel.

Conceptually, a program interpreter can examine each expression and decide whether or not to offload that single expression onto a remote node or to evaluate it locally. From the programmer's point of view there is no difference between the two approaches the program interpreter takes - they both result in the same answer.

6.1 Language Grammar

There are numerous challenges in implementing a partial shared-memory program; questions that sprang to mind upon conception of a partial shared-memory approach were numerous. For example, how would a changing variable be reflected to other threads/processes? How does the interpreter determine if a function $F()$ is available on a different thread/process of execution? How can the interpreter determine if it is safe to offload function $G()$ to a separate node. What if $G()$ calls $F()$ in some indirect fashion and $F()$ is not available on the node that $G()$ is executing on? What if both $G()$ and $F()$ are executed out-of-order due to running on different nodes?

To address many of the more obvious issues the approach chosen was to prefer a functional language over an imperative one. This allows the expression evaluator to assume that functions have no side-effects and to allow expressions to be evaluated out-of-order, mitigating the most challenging issues in the implementation of a partial shared-memory program.

One advantage of using a Lisp based grammar and syntax for the input is the relative simplicity of tokenising a series of parenthesised prefix expressions into a tree that can be evaluated. Another advantage with a regular and consistent syntax is that very few exceptions to a general rule need to be written. Using an imperative style of input would have required special keywords or grammar to define variables or specify operation order. A parenthesised prefix expression is flexible enough to represent both variable definitions and operation order with the same simple rule - *All expressions are surrounded by parentheses and the first element of the expression denotes the operation.*

This allows a single type of expression to be used no matter the operation needed.

For example, representing a calculation can be as simple as `(+ a b)` with the `+` being the operation, and defining a variable will be identical save for the operation name - `(define symbol value)`.

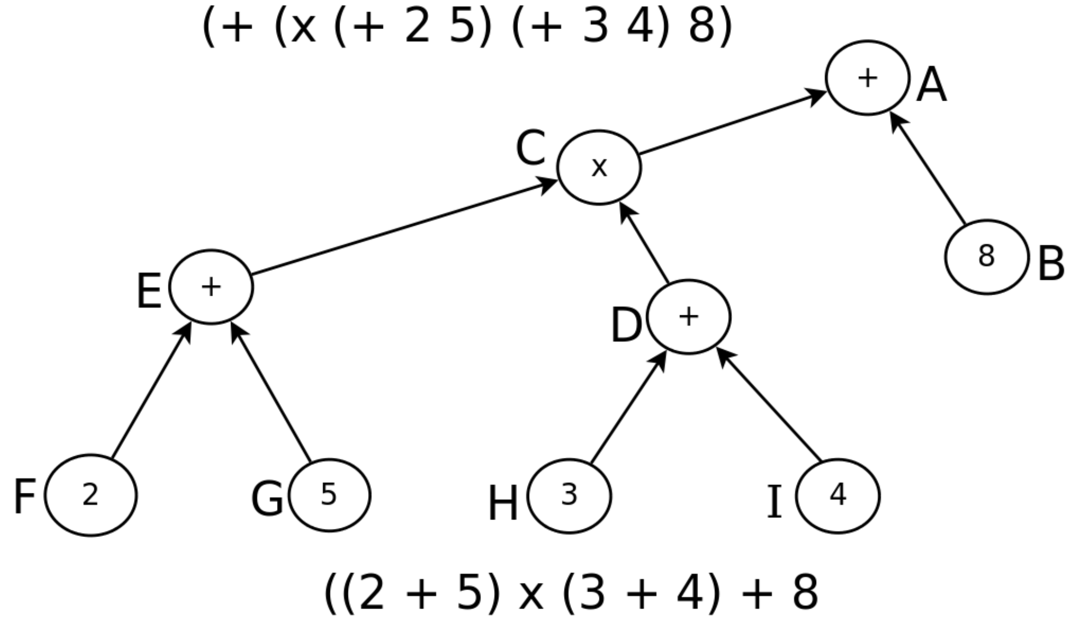

Figure 6.1: *Storage of a simple expression*

However the most compelling reason for using parenthesised prefix expressions over, for example, infix expressions, is the ease with which a parenthesised expression can be transformed into a tree with branch-nodes specifying operation and leaf-nodes specifying values. As can be seen in *Figure 6.1* the prefix expression can be used to construct a tree in a single pass from left to right while the equivalent infix expression will need multiple passes.

The final item of note is that parenthesised prefix expressions are capable of representing a turing-machine equivalent computation given the necessary primitives*[Boyer and Moore, 1983]*. While these primitives are missing in the expression evaluator developed over the course of this investigation, they are trivial to add to the existing codebase.

For example, the interpreter provides a `p-define` operator which defines a variable on a logical program stack. This can be easily changed to instead define a variable within another list, and subsequently use that list as a temporary program stack (an `alist`, in Lisp terms, which is an associative array) which is used for symbol resolution for the subtree beneath the `p-define` branch-node.

Another example would be defining new functions in terms of the existing language, i.e. creating new functions not in *C* but in the parenthesised prefix-notation language used by the interpreter. Because there is already an existing primitive which merely returns all of the evaluated child-nodes as a single list, functions can be defined in terms of subtrees with `alist`'s containing closures over the subtree to hold the parameters to the function. The addition of a closure allows the interpreter to implement recursion within any particular portion of the tree.

What that all means, on a practical level, is that although the expression evaluator is built as a proof-of-concept arithmetic evaluator, the tree traversal (which is the process that gets parallelised over several computers) can represent any computer program, not just arithmetic expressions, given the required primitives.

Any parallelism implemented via tree traversal applies to computer programs in general, not just to the evaluation of arithmetic expressions.

6.2 Expression Evaluation

Once a tree is constructed, a simple recursive traversal and application of branch-node operations onto leaf-node values is sufficient to evaluate the expression. A tree is composed of branch-nodes (signifying an operation) and leaf-nodes (signifying a value). A branch-node may possess one or more child-nodes, and a child-node may be either a leaf-node on the tree (i.e. a terminal node) or a branch node (i.e. consisting of one or more child-nodes).

The evaluation process is as follows:

- Evaluation of a leaf-node results in a value.
- Evaluation of a branch node results in all of the child-nodes attached to that branch being evaluated.
- Upon completion of a branch-node evaluation, the branch-node is replaced with the result of its evaluation and becomes a leaf node.

The evaluation process is depicted visually *Figure 6.2.*

The ability to evaluate an entire expression in parallel is natural with a tree such as in *Figure 6.1.* An example of parallel tree traversal with two computers, a *Red* computer and the *Green* computer, is shown in *Figure 6.3.* The tree is broken at two nodes, *Node E* and *Node D.* The subtrees consisting of *Node E* is then evaluated on the *Green* computer while the subtree consisting of *Node D* is

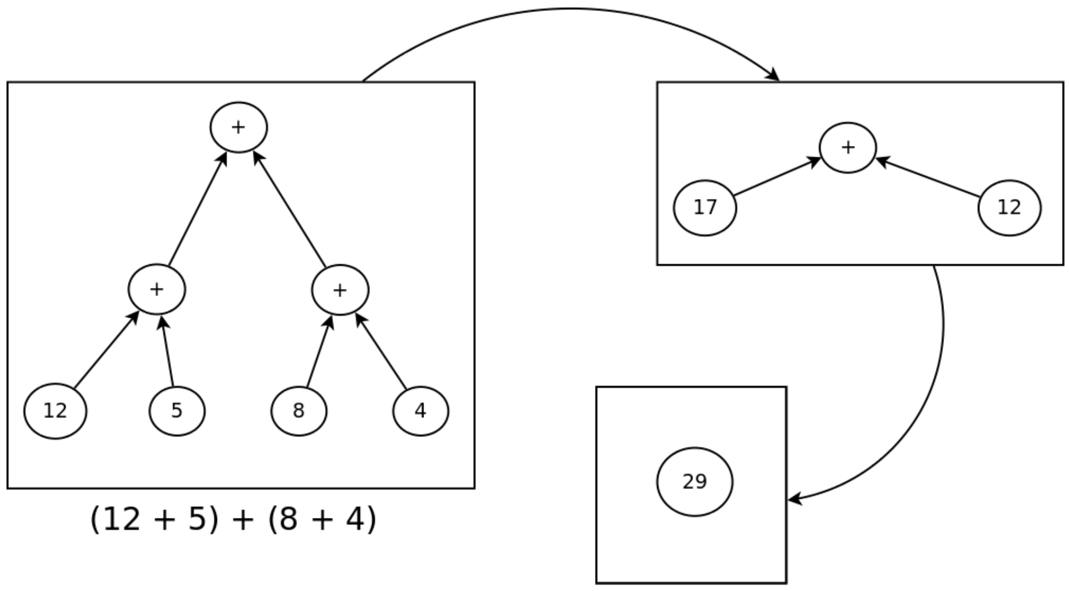

Figure 6.2: *Sample evaluation of a simple expression*

evaluated on the *Red* computer as shown in *Figure 6.4*. Once the evaluations on each computer is completed those branches in the local tree are then replaced with leaf-nodes containing the results from the *Red* and *Green* computers and evaluation via traversal continues as normal.

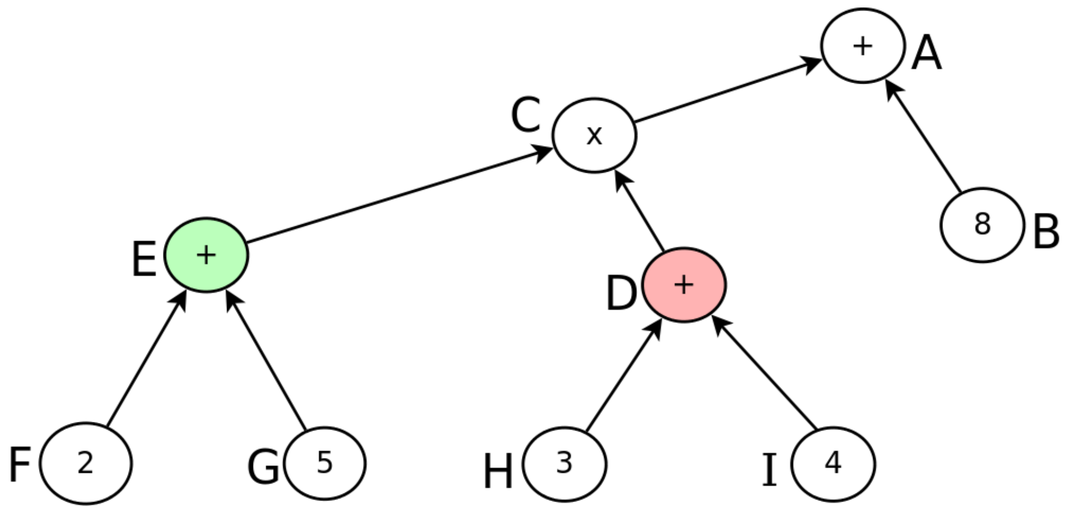

Figure 6.3: *Parallel evaluation: Break off entire branches for each computer to process*

It should be noted that performances in using a tree-traversal method in evaluating expressions would only be gained if the subtrees selected for parallel evaluation were of a sufficient level of computational complexity. There are currently no methods to determine the duration of an execution before it starts (or even if an execution will ever end)*[Hehner, 2013]*, however there are methods available that use past execution of a problem to estimate duration of the next execution*[Saha*

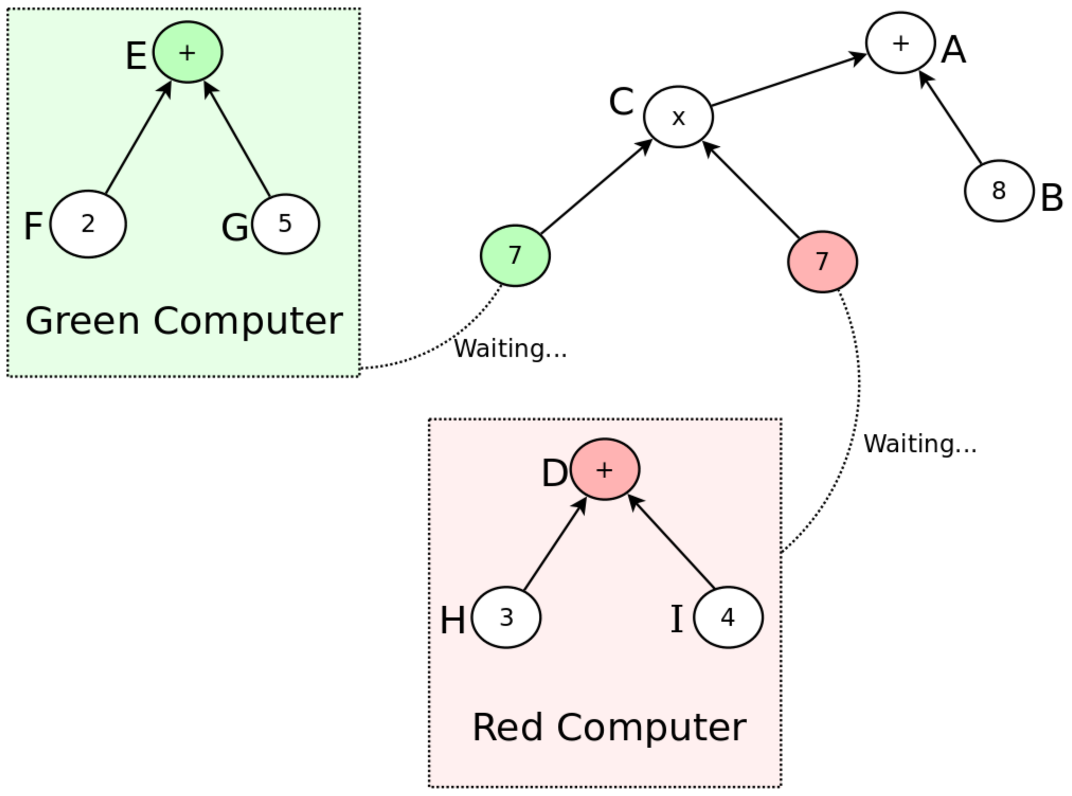

Figure 6.4: *Parallel evaluation: Wait for each computer to return a result*

et al., 2013].

6.3 Shared and Non-shared Memory

As variables and variable referencing is supported within the interpreter there still remained the problem of how to determine if a branch-node in the tree referenced a variable that did not exist on a remote computer. Two methods were devised to deal with this challenge:

1. A subtree which is transmitted to a remote computer for evaluation is transmitted along with a copy of the program stack that contained only the referenced variables.
2. During tree construction any variable references (symbols) were looked up immediately and substituted with the actual values if possible.

After implementation *Method 1* from above was deemed superfluous due to the efficiency of *Method 2*. Distinctions between operations safe for remote-execution and operations unsafe for remote-executions were programmed into the system. This ensured that any branch-node operator such as the assignment of a variable would

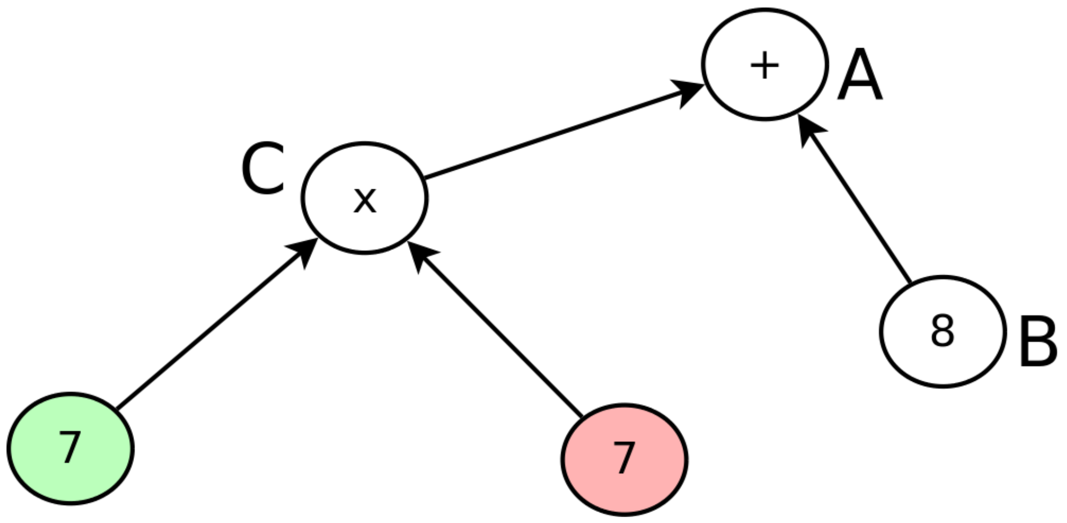

Figure 6.5: *Parallel evaluation: Resume normal traversal*

not be executed remotely.

It quickly became clear that memory does not need to be shared between the parallel executions that occurred during tree traversals - the process explained in *Section (6.2)* proved simpler than expected in practice. Any time a value is needed by a parallel execution of the tree traversal a copy of the value is sufficient to complete the parallel evaluation.

While copying of values were sufficient for evaluation, there was still the problem of *value assignment* using the operator `p-define`. Any subtree being evaluated may contain a branch-node that uses the `p-define` to assign a value to a named-variable. For example,

```
(p-define max 12)
```

assigns the value of 12 to the variable named `max`. Additionally, the expression representing the value might not be a literal - it might be the result of an expression itself. For example the following expression assigns the result of a subexpression to the variable `max`

```
(p-define max (* (+ 2 4) (- 3 4)))
```

Consider the problems in the above expression. The first problem is that the expression is parsed left-to-right. This means that the variable is encountered prior to any definition of it. The tree construction cannot perform a substitution of `max` with the actual value of `max` because `max` does not exist at that point.

The next problem is that the expression specified as the value to use must first be evaluated before `max` can be assigned.

Both these problems encouraged a re-examination of the tree-construction method. A solution was implemented such that the construction of the tree did not attempt any variable lookups. In brief, name-resolution of nodes with a symbol and not a value was deferred to just before evaluation. This allows the variable to be ignored until such time that all the dependent evaluations were completed.

The evaluation proceeds recursively and begins with the leaf-nodes of the expression. This means that by the time the assignment operation is needed the evaluation of the expression is completed, regardless of whether or not the interpreter evaluated the subexpressions locally or remotely.

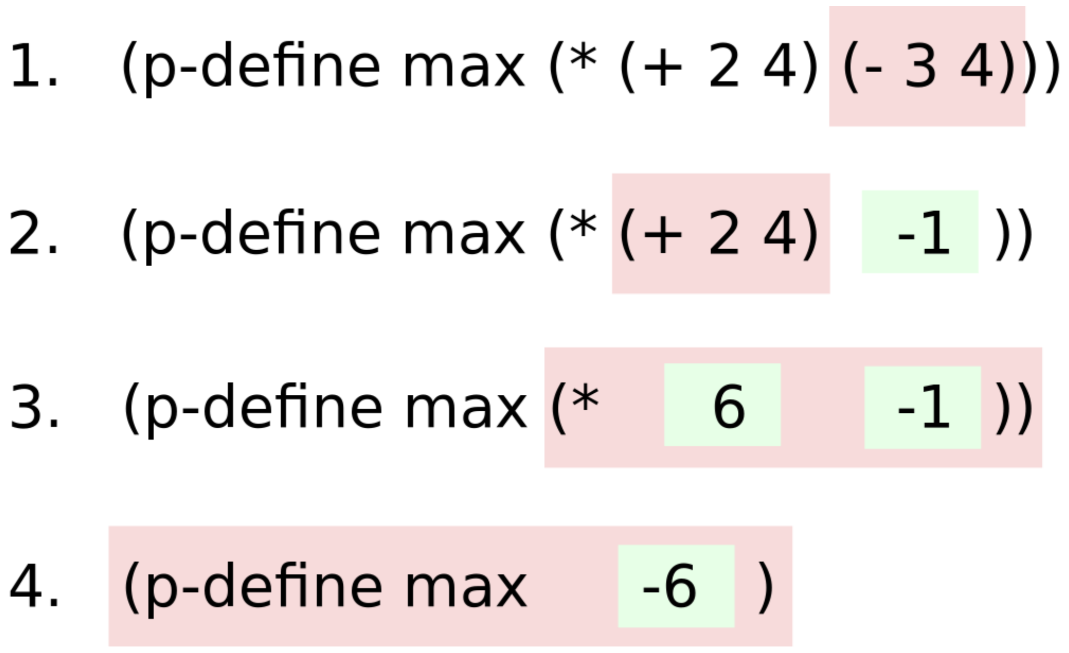
1. (p-define max (* (+ 2 4) (- 3 4)))
2. (p-define max (* (+ 2 4) -1))
3. (p-define max (* 6 -1))
4. (p-define max -6)

Figure 6.6: *Assignment Evaluation step-by-step*

An example of each step of the evaluation is shown in *Figure 6.6.* Each of the expressions in each step that are due for evaluation are shaded in pink while the results of the evaluation of the previous step is shown shaded in green. This reinforces the earlier justification for choosing a parenthesised prefixed-notation expression format.

6.4 Summary

The goal of this investigation is considered to be achieved: it is possible to perform multiprocessing using only remote-function invocations and without a *shared-memory* model nor a *message-passing* model.

Traditional *shared-memory* in the context of multi-threaded and multi-processing environments refers to a variable visible to all threads of execution. Changes to this variable is achieved using *Mutexes* (Mutual exclusion locks)*[Wong et al., 2014]*. A thread that shares a variable with another thread in this manner expects to lock the variable for exclusive access any time either thread accesses that variable, regardless of access type (read/write). The sharing of a variable under a *shared-memory* scheme means that the variable is shared in both directions.

The approach chosen for the expression evaluator developed in this chapter is a *partial shared-memory* approach. This differs markedly from the traditional *shared-memory* approach in that the data that is "shared" between different computers are shared in one direction only. The sharing is not reciprocal. This leaves the implementation the option to send only copies of the data to the destination thread when data is shared, as the source thread (the one hosting the variable) is guaranteed one-way sharing only.

The traditional *shared-memory* approach was found to be largely unnecessary to represent parallel tree evaluators. Explicit message-passing was likewise unnecessary and the parallel evaluation of subtrees occurs in a manner fully transparent to the user or programmer.

In the implementation of *partial shared-memory* the subtrees were shared over a network with remote computers. An implementation performing similar sharing between threads on a local computer in a shared address space would probably benefit from repurposing the *Copy-on-Write* model usually found in computer-disk related algorithms*[Wu et al., 2015]*. Trees are unusually well-suited to a *Copy-on-Write* model as subtrees that do not differ between copies can be reused trivially by merely reassigning pointers.

There is potential to use the existing developed system as a step towards a turing-complete interpreter*[Boyer and Moore, 1983]*. Such an implementation would involve the addition of temporary stacks to store variables. Creation of stacks (to store variables local to a subtree) was originally planned but found to be superfluous, although remnants of subtree stacks can be found in the codebase still (see *Appendix B*).

Much like existing Lisp implementations, the definition of a function would be simply a subtree, and as a subtree it can then be treated identically to every other variable. A function definition subtree can be attached to any particular point on the tree prior to traversal, which means that portions of the tree that get evaluated on remote nodes will automatically retain all context necessary for complete evaluation.

No need was found for *shared-memory*, nor for programmer-managed or explicit message-passing. All that is necessary is the ability to transmit subtrees to remote computers, and then evaluate them on those remote computers.

As intended, parallel computation is achieved with neither the need for shared-memory, nor the need for explicit message-passing.

Chapter 7

Findings

7.1 Objectives

The bulk of this thesis dealt with the technical deliverables needed to achieve the objectives listed in *Section (1.3)*. The most important objective, *"Develop a remote function-execution mechanism"* was detailed in *Chapter (3)*. *Rexel* was developed as the *Remote Function Execution Mechanism*, and has met all the goals needed to achieve the other objectives.

This mechanism was then rigorously tested against an industry standard competitor, *MPI*, to determine a comparative performance figure for *Rexel*. The methodology, testing, analysis and results of the benchmarking is thoroughly documented in *Chapter (4)*. The findings of the analysis indicated that while *MPI* possesses a slight performance advantage to *Rexel* within certain specific parameters, *Rexel* did indeed hold a performance advantage within other (fewer) specific parameters.

Overall, *Rexel* performed considerably better with regards to consistency of performance even when performing within those test parameters in which *MPI* performed better. *Rexel* was more consistent, more of the time. The full summary of the benchmarking findings are presented in *Section (4.8)*.

The advanced objective, *"Distributing the load of unmodified single-threaded, single-user and single-processor binary program on a cluster"*, was achieved by executing an unmodified *SQLite* program within a *Rexel* enhanced environment in such a manner that all nodes in the cluster were used. The development and results of this experiment are discussed in detail in *Chapter (5)*.

Although the objective was achieved, the results were mixed and considered mostly successful. The effort needed to turn the experiment into a fully successful one is documented as the results of the experiment in *Section (5.3)*.

The last advanced objective comprised a large part of this project; the objective was to determine if remote function execution leads to any new software architecture and/or design pattern. The goal of this objective was to determine if it is possible to develop a parallel execution model that required neither *shared-memory* nor *message-passing*.

The program written to investigate this interesting avenue is an evaluator for arithmetic expressions. The program was written to read prefix-notation arithmetic expressions, evaluate the expression and print the answer. The conclusions of this investigation is presented in *Section (6.4)* with the investigation itself document in full throughout *Chapter (6)*.

In brief, the investigation into a *partial shared-memory* approach to program design found that the expression evaluator worked exactly as intended by offloading subexpressions onto remote nodes without requiring the input grammar to be capable of expressing parallel constructs. The *partial shared-memory* approach resulted in values being shared in one direction only (partially shared) thus doing away with the need for mutexes on the memory being shared.

Two extra points of interest are also presented in *Section (6.4)*:

1. With very little effort the evaluator can be turned into a full turing-complete machine, making it possible to develop general purpose software that uses *partial shared-memory* within an expanded evaluator, and
2. There is the possibility of further optimising the evaluator to execute concurrently on a single machine using *Copy-on-Write* to implement the partial sharing of values.

The final conclusion of *Chapter (6)* is that parallel computation was achieved across multiple computers requiring the use of neither *shared-memory* nor *message-passsing*.

7.2 Research Questions

The completed objectives, both core and advanced, lead to a few not fully unexpected answers to the research questions and a few very surprising answers.

As expected, there are few mechanisms supporting the execution of remote functions; the lower layers needed to transmit, load and execute a function already exist in the form of system calls like `dlopen()` and `dlsym()` to load functions at runtime, and libraries such as `libffi()` to execute undeclared functions at runtime. However, until the development of *Rexel* pulled these tools together into a single coherent system, these were disparate tools that had little relation to each other.

Also not totally unexpected is the finding that while there are measurable performance differences between a single program that runs on multiple computers and multiple programs running on multiple computers, these differences are small. The comparative testing performed in *Chapter (4)* displayed an advantage to the *MPI* solution in terms of performance and an advantage to the *Rexel* solution in terms of consistency.

The finding that the existence of remote function execution as exemplified by *Rexel* did in fact open up new possibilities for program design was also somewhat expected, as this was the goal all along. The unexpected benefits came as somewhat of a surprise even though preparation was made to exploit new opportunities.

For example, it was gratifying to discover that *Rexel* could provide parallel execution functionality even to existing and unmodified software, as presented in *Chapter (5)*. As emphasised in *Section (5.3)*, though, there is more benefit in utilising *Rexel* directly from within the source code of software rather than by wrapping the unmodified program within a rexel environment.

The expression evaluator *Chapter (6)* was initially designed to exploit remote function execution by offloading subexpressions onto remote nodes. Development of this evaluator resulted in the realisation that the specific opportunity being exploited (*partial shared-memory*) could apply to parallel computation on individual computers.

Further, it was realised towards the end of the development effort that the software could be turned into a proper general-purpose programming language while *still* exploiting the new possibility of *partial shared-memory* that was opened up. This was an unexpected development and, had this been known at the start of the development, more effort would have been expended exploiting this singularly interesting discovery.

Even more surprising were the possibilities that were investigated but not included in this report (a few are mentioned in *Appendix D*), such as the ability to process suspect data on a remote machine (such as during input-randomisation testing*[Avgerinos et al., 2014]*).

Overall, *Rexel* has demonstrated that:

- Mechanisms to execute native-code functions remotely do exist, and
- A remote function invocation model can replace a *shared-memory* model and a *message-passing* model, and
- There are few measurable differences between executing a single program over multiple computers vs executing multiple programs on multiple computers, and
- The effects of a remote function execution capability on software design is both dramatic and positive, resulting in multiple new opportunities. These opportunities were all too numerous to exploit within the scope of this thesis, although the more interesting possibilities were pursued, demonstrated and reported on.

7.3 Further Avenues of Research

The findings produced by this research effort raise many more questions. For example, how does the comparative performance profiles as presented in *Section (4.8)* change for both the approaches chosen when the test parameters are increased? Are there any performance optimisations possible in *Rexel* that would match the 20 years of performance optimisations and refinements in *MPI*? At what point does the extra effort and complexity of *MPI* exceed the performance benefits gained?

Chapter (6) raises many more questions. Is an effort to enhance the software to be turing-complete a good expenditure of effort (does the world need another programming language, even if it does bring something new to the table)? As a general-purpose software language, would the expression evaluator result in even more new opportunities that are made possible with a *partial shared-memory* approach? Can the software be optimised to selectively parallelise code-snippets based on complexity measures?

This thesis exploits only the most obvious possibilities of a *Rexel*-type framework; no doubt there are many more that exist which are yet to be discovered.

Bibliography

Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 33–42. ACM, 2012.

Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

Robert S Boyer and J Strother Moore. A mechanical proof of the turing completeness of pure lisp. Technical report, DTIC Document, 1983.

Richard Brown, Patrick Garrity, Timothy Yates, MN Northfield, Elizabeth Shoop, and MN Saint Paul. Teaching map-reduce parallel computing in cs1.

Valeria Cardellini, Alessandro Fanfarillo, and Salvatore Filippone. Overlapping communication with computation in mpi applications. *Universita di Roma Tor Vergata, Tech. Rep. DICII RR-16.09*, 2016.

Günther Charwat, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Andreas Pfandler, Christoph Redl, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. Vcwc: A versioning competition workflow compiler. In *Logic Programming and Nonmonotonic Reasoning*, pages 233–238. Springer, 2013.

M Clemencic and B Couturier. A new nightly build system for lhcb. In *Journal of Physics: Conference Series*, volume 513, page 052007. IOP Publishing, 2014.

Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable memory use in mpi: A case study with mpich2. In *EuroMPI*, pages 140–149. Springer, 2011.

Anthony Green. The libffi home page. *URL http://sourceware. org/libffi.*

Eric CR Hehner. Problems with the halting problem. *Advances in Computer Science and Engineering*, 10(1):31, 2013.

Lorin Hochstein. *Ansible: Up and Running.* " O'Reilly Media, Inc.", 2014.

Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95(12):1121–1136, 2013.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL*, volume 96, pages 295–308, 1996.

Pradeeban Kathiravelu and Luis Veiga. Concurrent and distributed cloudsim simulations. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 490–493. IEEE, 2014.

Brian W Kernighan and Phillip James Plauger. The elements of programming style. *The elements of programming style, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978.*, 1, 1978.

Duncan Temple Lang. Rffi: Interface to libffi to dynamically invoke arbitrary compiled routines at run-time without compiled bindings.

Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

Andreas Löscher and Konstantinos Sagonas. The nifty way to call hell from heaven. In *Proceedings of the 15th International Workshop on Erlang*, pages 1–11. ACM, 2016.

Dejan Mitrović, Mirjana Ivanović, and Zoran Budimac. Erlang and scala for agent development. 2013.

Kato Mivule, Benjamin Harvey, Crystal Cobb, and Hoda El Sayed. A review of cuda, mapreduce, and pthreads parallel computing models. *arXiv preprint arXiv:1410.4453*, 2014.

mpi forum.org. Mpi standard v3.1. ”`http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`", June 2015. [Online; accessed 19-Aug-2016].

Kaan Onarlioglu, Mustafa Battal, William Robertson, and Engin Kirda. Securing legacy firefox extensions with sentinel. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 122–138. Springer, 2013.

John Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.

Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for smt processors, 2000.

Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Thomas Anderson, Arvind Krishnamurthy, Mark Zbikowski, and Doug Woos. Towards high-performance application-level storage management. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

Martin Rinard. Analysis of multithreaded programs. In *International Static Analysis Symposium*, pages 1–19. Springer, 2001.

Rupsa Saha, Shweta Rai, Srikanth Prabhu, and M Geetha. Determining effectiveness of multithreading for solving problems with low computational complexity. *International Journal of Computer Science & Communication*, 4(2):161–166, 2013.

Kenneth C Smith, Alice Wang, and Laura C Fujino. Through the looking glass: Trend tracking for isscc 2012. *IEEE Solid-State Circuits Magazine*, 4(1):4–20, 2012.

Milan Stevanovic. Linux toolbox. In *Advanced C and C++ Compiling*, pages 243–276. Springer, 2014.

Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3 (7):54–62, 2005.

Michael Bedford Taylor. Bitcoin and the age of bespoke silicon. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 16. IEEE Press, 2013.

Tom White. *Hadoop: The definitive guide.* ” O’Reilly Media, Inc.”, 2012.

Michael Wong, Eduard Ayguadé, Justin Gottschlich, Victor Luchangco, Bronis R de Supinski, Barna Bihari, et al. Towards transactional memory for openmp. In *International Workshop on OpenMP*, pages 130–145. Springer, 2014.

Xingbo Wu, Wenguang Wang, and Song Jiang. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 15. ACM, 2015.

Shuangyang Yang, Walter Ligon, and Elaine C Quarles. Scalable distributed directory implementation on orange file system. *Proc. IEEE Intl. Wrkshp. Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

Appendix A

Format of *Rexel* Configuration

Rexel configuration data for those programs using *Rexel* as a master is stored in a set of files on the machine that is to run the *master* component of *Rexel*. The client program need not be written specifically to load the configuration files - a constructor in `librexel-master.so` automatically loads the configuration files when the library is loaded.

There are potentially multiple configuration files. Each of them may specify different nodes in the cluster, and configuration data from the files read later in the process will be preferred over the configuration data read earlier in the process.

Configuration data need not conflict, as the configuration files implement namespaces so that the same node may be specified multiple times in separate namespaces.

1. The file `/etc/rexel/rexel-masterrc` is processed if found,
2. The file `$HOME/.rexel-masterrc` is processed if found,
3. The file `$PWD/rexel-masterrc` is processed if found,
4. Finally the file `$PWD/.rexel-masterrc` is processed if found.

The format of the file is simply a single command on a single line. Each command takes a single argument. All of the example programs have a `rexel-masterrc` file to work with, so there are plenty of examples in the codebase.

The commands are:

start_list Create a new namespace, switching to it.

in_list Switch the namespace.

end_list End the current list (switch to no-list).

add_node Adds a node to the current list.

del_node Unimplemented, RFU

The format of a node added with `add_node` is `address:port`. Namespace names and addresses may not have spaces in them. Comments are started with a single '#' character and extends from that character inclusively to the end of the current line.

An example configuration file used during development of the sqlite3 wrapper libraries:

```
start_list netdup       # Create and use namespace 'netdup'
add_node 10.0.0.201:42424 # Add a node
add_node 10.0.0.202:42424 # Add a node
add_node 10.0.0.203:42424 # Add a node
add_node 10.0.0.204:42424 # Add a node
```

Upon initialising the Rexel-master API the caller specifies the name space containing the nodes they wish to use.

Appendix B

The Uncommon Lisp Environment

The Expression Evaluator developed for this project is based on a regular grammar that is loosely based on Lisp-like programming languages. This section deals with the details of implementing the interpreter, called *The Uncommon Lisp Environment* (abbreviated as *Uncle*).

B.1 Expression Evaluation

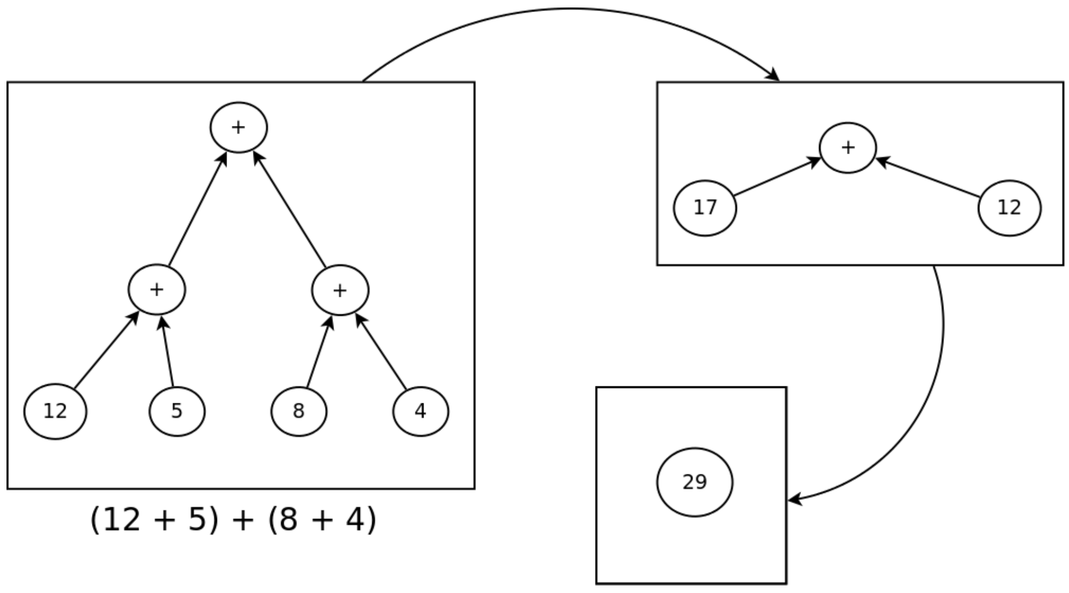

Figure B.1: *Sample evaluation of a simple expression*

All expressions can be expressed in terms of an evaluation tree. Traversing this tree in the correct order and replacing each node with the result of evaluating all that node's children is an easy way to visualise the evaluation of a single expression. A simple example showing infix expressions is depicted in *Figure B.1.* However, the design of the expression evaluator discussed in this section uses prefix expressions as input, not infix expressions.

Infix expressions have limitations. Specifically, the order of the evaluation has a significant effect on the result and certain rules have to be adhered to *even with* the presence of parentheses to clearly delineate subexpressions. However using either prefixed or postfixed expressions such as those in Lisp-like languages or stack-based languages removes all ambiguity from the expression.

As an example, the infix expression "$12{+}5{\times}8{+}4$" is ambiguous until parentheses are added, while the prefix expression "$\times + 125 + 84$" and the postfix expression "$125 + 84 + \times$" are both unambiguous and clear. The prefix expression is a natural fit for a tree structure. Even simple input such as "$((2 + 5) \times 4) + 8$" requires more than a single pass to evaluate, while the equivalent prefix expression "$+ \times 4 + 258$" can be evaluated as each operand is encountered. Ordinarily more than a single pass through an expression is acceptable, but the goal of the expression evaluator developed for this investigation was to offload subexpressions onto remote nodes. This requirement precludes forced-order evaluation of subexpressions.

Another reason for choosing prefix expressions over infix expressions is the relative ease with which they can be extended from *binary* operations to *n-arity* operations. Lisp-like languages (such as Common Lisp and Scheme) use this facility to great advantage, in the process simplifying the prefix representation. Using unlimited-arity, the expression "$+ + +25 + 48 + 36$" becomes a more understandable "+ 2 5 4 8 3 6". Subexpressions are, in this representation, parenthesised. The parenthesised version appears as "(+ 2 5 4 8 3 6)", and, as part of a larger expression (for example, finding an average), "(/ (+ 2 5 4 8 3 6) 6)".

Finally, the decision to use prefix expressions was influenced by a very practical concern, namely that such representations have in the past been extended into full programming languages. While it is perhaps unlikely that the expression evaluator developed in this section becomes a serious programming language, the facility for doing so comes at no cost when using prefix expressions.

For example, operators can be thought of as primitive functions. This means that functional programming can be added in the future with very few changes to the grammar used in the input data.

B.2 Design of Expression Evaluator

B.2.1 Evaluation

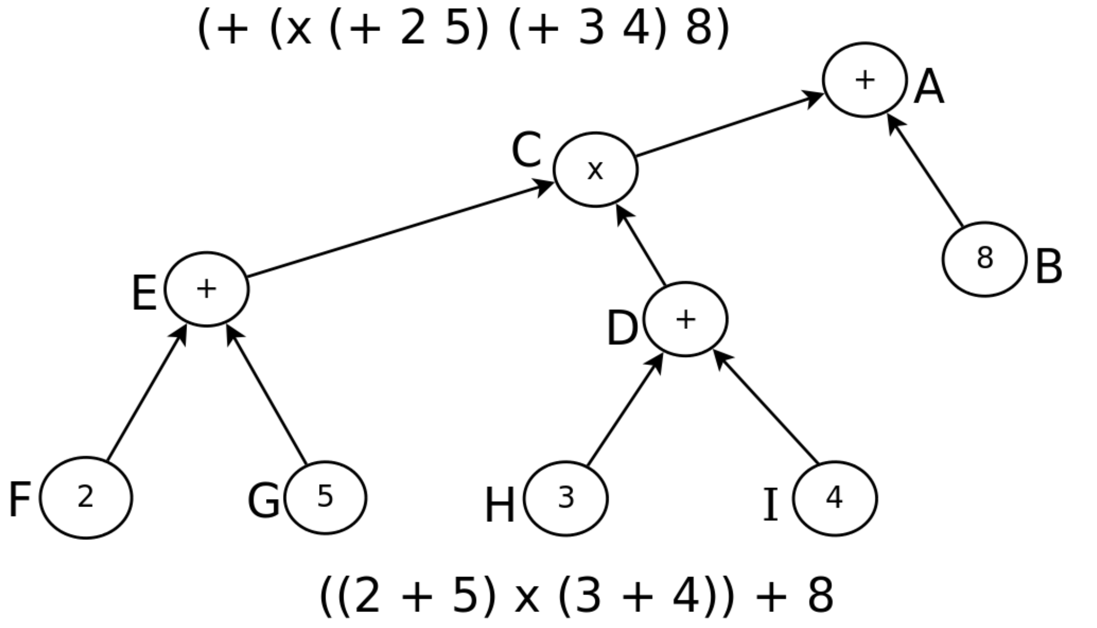

Figure B.2: *Storage of a simple expression*

All expressions can be expressed in terms of an evaluation tree. The approach chosen is similar to the current functional language approach used by systems such as *Haskell* in that it is an *"implicit, semantically transparent parallelism" [Jones et al., 1996].* Each element in the input is constructed as a node in a tree, with values (both variables and literals) stored as leaf-nodes and operators stored as branch-nodes as shown in *Figure 6.1.*

The tree is made of branch-nodes and leaf-nodes. Evaluating a leaf-node simply results in that leaf node. Evaluating a branch node causes a recursive evaluation of all that branch-nodes leaf-nodes. For example, evaluating the leaf-node "G" in *Figure 6.1* simply returns "5". Evaluating the branch-node "" cause evaluation of "E" and "D", which in turn causes the evaluation of "F", "G", "H" and "I".

Every branch in the tree can be independently evaluated without waiting for any non-child branch. For example the branch represented by $3 + 4$ can be evaluated without any results needed from the branch represented by $2 + 5$. As these two branches are not dependent on each other they can thus be evaluated in parallel.

In the implementation of the expression evaluator these branch-nodes are evaluated on different logical nodes in a cluster. Once a branch-node is fully evaluated it is replaced with a leaf-node containing the results of the evaluation. The process is then repeated for the parents of the new leaf-node, and in this way the entire expression is evaluated making use of as many remote computers as there are available, one for each branch-node evaluation.

From a users's point of view the multi-processing is fully transparent. The evaluation of each branch-node may take place on the local computer or (unknown to the user) the interpreter may offload a certain branch to a remote node.

While the interpreter possesses a program stack (in addition to the native code stack provided to all running programs by the *OS*), evaluation rarely needs to access the stack. The tree is populated with values as determined at time of tree construct, with few exceptions. This simplifies the processing somewhat, in that the interpreter never needs to moderate exclusive access to a single structure that is shared by all processes.

B.2.2 Variables

As variables and variable referencing is supported within the interpreter there still remained the problem of how to determine if a branch-node in the tree referenced a variable that did not exist on a remote computer. Two methods were devised to deal with this challenge:

1. A subtree which is transmitted to a remote computer for evaluation is transmitted along with a copy of the program stack that contained only the referenced variables.
2. During tree construction any variable references (symbols) were looked up immediately and substituted with the actual values.

After implementation *Method 1* from above was deemed superfluous due to the efficiency of *Method 2*. Distinctions between operations safe for remote-execution and operations unsafe-for remote executions were programmed into the system. This ensured that any branch-node operator such as the assignment of a variable would not be executed remotely.

Due to the functional nature of the expression-evaluator it is neither necessary nor desirable to have more than one operator that can access the program stack. Branch-nodes, when evaluated, return the results - they do not perform any intermediate storage of the result and they do not need to.

There are exceptions made; a single primitive operator is provided to add or replace a variable on the stack. Other than this isolated operator, all other operators do not use the program stack as the variable value substitution is performed prior to any evaluation taking place.

Unlike many other functional programming languages, variables are mutable although a diagnostic is generated each time a variable is changed. For all practical purposes in its intended usage variables are only assigned once, and rarely would a user need to reassign a variable's value unless they intended to simply reuse the variable for some other purpose.

B.2.3 Tree Construction

Although the input for the expression evaluator very much resembles the syntax of Lisp, there are large differences. For example, Lisp uses a table at run-time (called the read-table) to attach semantics to input tokens. The *Uncommon Lisp Environment* does not do so and uses a compile-time table generated using a lexical scanner, *Flex*.

The scanner recognises the start of a branch, *"("*, the end of a branch, *")"* and leaf nodes. Each token is either a number, a symbol or a string. Numbers are all of type `double` while both symbols and strings are NULL-terminated character arrays. Strings are differentiated from symbols by enclosing double-quotes.

Each token read in is stored in an atomic type, `atom_t`, which is then attached to the current branch. The token signifying the start of a new branch causes a new `atom_t` of type branch to be created. This new branch is attached to the current branch, then this new branch is made the current branch and processing of input tokens resumes.

Once a token signifying the end-of-branch condition arrives in the input stream, the current branch is ended, and its parent is made the current branch. This simply yet flexible mechanism allows the interpreter to build complex trees of arbitrary length.

Once the tree is fully constructed the main program is notified that a new element is ready for evaluation. This element may be either a branch-node or a leaf-node, because it does not matter as both types of nodes are evaluated via the same function. The main program then passes the element to a pre-evaluation function which substitutes all variable references with actual nodes. This step is important because the operators themselves are variables.

Thus, when the symbol “+” is encountered, the constructor of the tree does not regard it as anything special - it is simply another variable reference. Prior to evaluation, when variable references are being resolved, the symbol “+” gets resolved like every other variable to reference on the stack.

The code implementing all of the above `token.yy` in the git repository.

Appendix C

SQLite Development

C.1 Input script used for the experiment

```
DROP TABLE test_one;

CREATE TABLE test_one (
   col_a INT PRIMARY KEY,
   col_b INT
);

INSERT INTO test_one VALUES (1, 3);
INSERT INTO test_one VALUES (2, 6);
INSERT INTO test_one VALUES (3, 9);
INSERT INTO test_one VALUES (4, 12);
INSERT INTO test_one VALUES (5, 15);
INSERT INTO test_one VALUES (6, 18);
INSERT INTO test_one VALUES (7, 21);
INSERT INTO test_one VALUES (8, 24);
INSERT INTO test_one VALUES (9, 27);
INSERT INTO test_one VALUES (10, 30);
INSERT INTO test_one VALUES (11, 33);
INSERT INTO test_one VALUES (12, 36);

SELECT * from test_one;
```

Appendix D

Potential New Opportunities

D.1 Fine-grained Unit-testing of functions

During development a surprising benefit of *Rexel* was encountered: functions in the running program which crashed due to bad input or bugs crashed on the remote machine *while leaving the main program intact.*

As all the remote function calls are implemented with timeouts, sooner or later the main program timed out on a function call and returned a timeout status. These occurrences naturally lead to developing this characteristic into an actual program.

The intention is that unit-testing can continue even in the cases of fatal bugs, as the main program is on a different computer to the crashing function. The program developed, `func_tester`, is in the git repo, currently in a very limited form but future enhancements are planned due to the remarkable utility of the program.

What `func_tester` does is allow the user to specify a library filename and a function within, along with a prototype to use for determining parameters for the function. The program then runs this function on *Rexel* remote nodes with random input a specified number of times, and produces a report providing the user with the input that caused the function to timeout.

D.2 Quorum-based problem-solving

There was not enough time to flesh this idea out with a demonstration via a small example program, nevertheless this could become a viable implementation of quorum-based problem-solving in disciplines where accurate results are critical. Many embedded systems, should they fail, place humans in danger; this is particularly true of the medical devices industry, the mining industries and many manufacturing industries.

With *Rexel* being compatible with a minimal *Linux* distribution, requiring nothing more than the kernel and a minimal set of system libraries installed, it should be possible to allow major decisions taken by these embedded computers to follow a quorum-based solution, in which all of the various computers forming part of the quorum have to get the same result of critical function calls before any potentially dangerous action proceeds.

As a followup, a paper is planned which uses Raspberry Pi compute modules in a set to execute critical software using *Rexel* to implement the quorum.

Appendix E

The Problem with Threads

E.1 Nondeterminism Case-study

Threads are notorious for non-determinism*[Lee, 2006]*. Threads can be deadlocked (causing a total halt in execution), can be starved (causing a partial halt in certain services) or prone to either under-locking (leading to a data-race condition) or over-locking (leading to a severe and non-obvious degradation in performance).

The implementation of *Rexel* faced numerous challenges with synchronisation when threads were used to implement the `rexel-node` program. Around 150 hours were spent tracking down intermittent bugs that were difficult, and in at least one case, impossible to reproduce. A switch to a fork()-based `rexel-node` architecture got the project back on track in less than 20 hours.

In advance of any criticisms aimed at the developer's competence (or lack thereof), there is at least one highly-cited paper detailing the problems with threads. The paper *[Lee, 2006]* highlights just how difficult it is to get threads correct. The *Ptolemy System*, having been developed by expert PhD's and multiple graduate students, along with experienced industry experts, a rigorous test schedule with 100% code-coverage, formal review processes, comprehensive test-units and formal proofs of correctness deadlocked after four years of operation.

This highlights the non-determinism inherent in *shared-memory* multiprocessing. It's not a question of *if* the program will crash, it's a question of *when* the program will crash. With threads in a non-trivial program, the system is almost certain to crash. It may not do it when you first deploy it, it certainly won't crash when you are unit testing it, but rest assured that it *will* one day halt, crash, deadlock, starve or data-race.

A rather older paper, *[Ousterhout, 1996]*, points out that some percentage of developers are capable in high-level languages with niceties such as garbage collection and no pointers, a smaller percentage of developers are capable with lower-lever languages like *C*, while an even smaller percentage is capable with complex languages like *C++*. The very smallest percentage of developers will be capable of safely using threads.

Which brings us to the all-important question: is your project employing the top 10% or top 5% (or top x%, for a suitably impressive value of x%) of developers? Because if you cannot honestly answer that question, then perhaps threads are not a good fit for your project.

Threads are still used within *Rexel*, but in a limited and *very* highly curated manner - I hesitate to allow proliferation of threads within code, keeping in mind the pearls of wisdom from one of the titans of Software Development history*[Kernighan and Plauger, 1978]*:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> *- Brian Kernighan*

E.2 Contention

At first glance threads seem a viable solution. A single program can easily take advantage of multiple cores and run multiple executions at once. However there is a limit to what threads can achieve. In a shared-memory threaded program, in which all threads share the same memory space and can access each others variables, the overhead of using threads needs to be balanced against the advantage of multiple executions.

In a program with threads, access to shared variables must be protected by a *Mutual Exclusion Lock* (mutex). Before any access is allowed each thread will attempt to acquire the lock. If the lock is busy then the thread is put into a queue to wait for the lock to become available again.

As a solution becomes more parallel threads become less viable due to the mutex waiting involved.

For a single threaded program the probability of acquiring exclusive access to a shared variable without waiting is 1. In a program with two threads the probability of either thread acquiring the lock without queueing is $\frac{1}{2}$ or 0.5. For a program with three threads that probability drops to $\frac{1}{3}$ (0.333). For a program with n threads the probability of a particular thread acquiring a shared resource without queueing is $\frac{1}{n}$.

As the number of threads grow the probability of not joining a queue when accessing a shared variable grows asymptotically towards zero. For a mere 64 threads the probability of not joining a queue is $(1 - \frac{1}{64})$ or 0.016, which means that a thread has a 98.4% chance of joining the queue.

Most practical programmers would argue that this isn't a problem if the correct design pattern is used (producer/consumer, etc). However, regardless of the design pattern used, if *any thread* writes to the shared-variable then *every other thread* has to first attempt to lock it exclusively before use. Of course if no thread writes to the variable then each thread can simply keep their own copy and there will be no contention for the variable. Most processors, in any case, simply employ caching strategies to reduce the constant access time of contended variables.

Regardless of the number of physical cores on the computer and the mitigation approaches, threaded solutions very quickly run into the above contention problem. To address contention it may be desirable to run multiple tasks on a single machine rather than multiple threads in order to maximise use of the multiple cores. This approach does address the shared-variable/shared-memory problem as each task will have its own memory space and need not contend for access to the same variables.

However you run into the above problem once again, only at a lower level. Even though a machine may have multiple cores, each core executes instructions stored in memory. Each instruction must be fetched from memory before execution, and thus begins contention of the physical address bus. On a modern desktop computer the address bus may be composed of between 48 and 52 copper traces that lead to the memory. Before any task can execute, the cores themselves will find themselves contending for the main memory even before any program data is needed.

This is the same contention problem we find with threads. The only solution appears to be by removing contention for the address bus, i.e. having individual and distinct cores run tasks, each using their own uncontended address bus and uncontended memory. In short, having different physical computers run the tasks.