Communicating Sequential Processes in High Performance Computing

Alasdair James Mackenzie King

August 20, 2015

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2015
Abstract

Communicating Sequential Processes has been used in High Performance Computing with success and offers benefits in providing a model for safe parallel and concurrent programs. Its mathematical basis allows for programs to be evaluated and proved for their completeness, insuring the program is free of common problems found within parallel and concurrent programming. Modern parallel programs are mainly implemented in MPI or OpenMP. These tools have had a long time to mature and provide programmers with very high performing programs for distributed and shared memory. This project aims to use CSP with a high level language to evaluate the usefulness and ability of the model and higher level languages. Python is a popular language and its use within HPC is increasing. An implementation of CSP should provide Python with the ability to program for High Performance Computers, and this allows programmers to make use of the features Python brings. This project will use PyCSP as the bases of the CSP implementation and make modification where required to run the library on ARCHER. Graph500 will be used as a benchmark for implementation, comparing the PyCSP version with a C MPI version supplied by the Graph500 specification. This project results in a partial implementation of the PyCSP features that work on ARCHER using MPI. The final program performed poorer than the C version. This was to be expected from Python and parts of the program were improved though the use of scientific libraries, providing performance oriented data structures in C. Due to the absence of some of the features the program could not scale beyond one node for the final implementation of Graph500. However the CSP model proves to be reliable with no communication problems during implementation. Further work could see a higher performing library providing CSP for programmers to help design and simplify the constructor of parallel and concurrent programs.
# Contents

1 Introduction ................................................................. 1
  1.1 Communicating Sequential Processes ............................. 2
      1.1.1 Building systems in CSP ................................... 3
      1.1.2 Implementations .......................................... 5
  1.2 Python Language & Its use within HPC ......................... 7
      1.2.1 Python Language .......................................... 7
      1.2.2 Parallelism and Concurrency ............................ 10
      1.2.3 Python Limitations in HPC ............................... 11
      1.2.4 PyCSP ...................................................... 12
  1.3 Graph500 .............................................................. 14
      1.3.1 Specification ............................................. 14
      1.3.2 Graph Generation ......................................... 14
      1.3.3 Breadth First Search .................................... 16
      1.3.4 Implementations ......................................... 16

2 Python/CSP implementation .............................................. 17
  2.1 PyCSP Process Creation ............................................ 17
      2.1.1 Distributed PyCSP Processes ............................. 17
  2.2 PyCSP Inter-process Communication .............................. 19
      2.2.1 Alternative Solutions and Considerations .............. 22
      2.2.2 Syntax for modified Parallel and modified Channels .... 23
      2.2.3 Alts! (Alternatives) and Guards ....................... 23
  2.3 Multiple Readers and Writers of networked channels .......... 25
  2.4 Graph Generation .................................................. 27
  2.5 Breadth First Search ............................................. 28
      2.5.1 Design .................................................... 28
      2.5.2 Implementation .......................................... 30

3 Results ................................................................. 35
  3.1 Graph500 MPI Benchmark (C version) ............................ 35
      3.1.1 Test conditions .......................................... 35
      3.1.2 Graph500 Results (C version) ........................... 36
  3.2 Python/CSP Results Comparison .................................. 38
      3.2.1 Errors ..................................................... 38
      3.2.2 Test Conditions .......................................... 38
      3.2.3 Execution Time ........................................... 38
      3.2.4 Errors during results ................................... 38
      3.2.5 Length of Program ....................................... 39
3.2.6 Performance Pitfalls ............................................. 40

4 Conclusions ......................................................... 41

5 Further Work ......................................................... 43
   5.1 Multi Channels .................................................... 43
   5.2 Distributed Processes ............................................. 43
   5.3 PyCSP upgrade ..................................................... 43
   5.4 CSP implementations ............................................ 44
   5.5 CSP design in other parallel implementations .................. 44

6 Appendix .......................................................... 45
   6.1 Graph500 Makefile ................................................. 45
   6.2 bfs_simple.c listing .............................................. 46
   6.3 First implementation of GraphReader.py ....................... 49
   6.4 Improved implementation of GraphReader.py ................... 50
List of Figures

1.1 A vending machine that takes in a coin and returns a chocolate and returns to wait on a new coin ................................................................. 4
1.2 A vending machine that takes in a coin and returns a chocolate or toffee .... 4
1.3 Example process graph of 3 concurrent processes connected with point to point channels. [1] ................................................................. 5
1.4 For loop comparisons between Python, C, Fortan and Java ............................. 8
1.5 Python list comprehension ....................................................................... 8
1.6 Declaration of Classes and Functions in Python ......................................... 9
1.7 Python’s Lambda syntax ......................................................................... 9
1.8 Python decorator example ....................................................................... 10
1.9 An example of a send in Mpi4Py with each of the two types of sends. ...... 12
1.10 Sample GNU Octave graph generator supplied by Graph500 .................. 15

2.1 A PBS script to run a python job on ARCHER ........................................ 18
2.2 PyCSP Parallel functions for process instantiation. Docstrings have been removed for figure clarity ................................................................. 19
2.3 Modification to provide each MPI Rank with a process to run .................. 19
2.4 Example process graph of 3 concurrent processes connected with single channels ........................................................ ................. 20
2.5 Network channel example showing the reading end sink ........................ 20
2.6 Network channel example showing the writing end source ....................... 21
2.7 Distributed channel management for Distributed Processes ....................... 21
2.8 Physical view of distributed PyCSP processes with networked channels ..... 22
2.9 Amendments to Alternative.py to allow for MPI channels. ....................... 24
2.10 Modifications made to mpi_channels ....................................................... 24
2.11 PyCSP Graph Solver graph with MPI rank numbers ................................. 25
2.12 Updated channel end discovery .............................................................. 26
2.13 Changes made to mpi_channel send. Similar changes are made for recv .................................................................................................. 26
2.14 Python serial version of the sample graph generation ............................... 27
2.15 Vampier Trace of graph500 sample C/MPI ............................................ 29
2.16 Distributed Breadth-First Expansion with 1D Partitioning from [28] ........... 30
2.17 CSP diagram for PyCSP implementation of Breadth First Search graph solver ........................................................ .......................... 31
2.18 Graph Decomposer process ................................................................. 31
2.19 Breadth first search worker ............................................................... 32
2.20 Queue processes ............................................................................. 33
2.21 Main program entry point ................................................................. 33

3.1 Modified Makefile for Graph500 test .................................................... 35
3.2 Graph500 sample code PBS script for one node at scale 12 .................... 36
# List of Tables

1.1 Graph500 Problem Class definitions [2] .............................................. 14  
3.1 Graph500 results for one full node - 24 process- at each graph scale ........ 36  
3.2 Graph500 results for Scale 12 .......................................................... 37  
3.3 Graph500 results for Scale 19 .......................................................... 37  
3.4 Graph500 results for Scale 22 .......................................................... 37  
3.5 Graph500 results for Scale 26 .......................................................... 37  
3.6 PyCSP execution times for one node at Scale 12 ............................... 39  
3.7 Line counts for each implementation ................................................. 39  
3.8 Execution time with initial graphreader.py implementation ................. 40
Acknowledgements

I would like to thank my supervisor Dr Daniel Holmes for his support and advice, and family and friends for enduring me talk at length about the project. Thank you to the EPCC for allowing the time and resources to work on this project.
Chapter 1

Introduction

Programs for High Performance Computing (HPC) have traditionally been programmed in well-established programming languages for scientific computing. Primarily C and Fortran, these languages dominate the development of scientific software for high performance computing. Java, C#, Python have done use by some using these higher level languages to perform the same tasks as C and Fortran. Some of the reasons for this, are new programers being taught these languages are more commonly found amongst both programers and people in their chosen field. These languages are more common to general systems programming and are often the first languages that some people can encounter. Some academic staff may have little programming background, and have taken to these languages for there ease of use or are part of the scientific software, for example Matlab. The Python programming language has found some use within High Performance Computing and is readily used outside of HPC. The language is often taught as a beginners language, but it has plenty of library support for scientific computing.

Programming in parallel can often be challenging for some programers and depending on the level of expertise, interacting with threads directly - pthreads - or other implementations is often looked over in favour of using higher level abstractions, such as OpenMP or MPI. These abstractions move some of the complexity of process management and communications away from the programer, providing a sanders interface to help them build high performing parallel and concurrent systems. In doing so this reduces the barrier to entry and allows people to make better use of the hardware. MPI provides a single process multiple data view with exploits communication between processes of a large distributed system to parallelise work. OpenMP provides the programer with compiler tags, that allow the compiler to insert and optimise the relevant code. This makes better use of shared memory machines that provide a large number of concurrent threads. Poor performing parts of programs can be annotated to tell the compiler to brake the work up amongst theses threads and improve the performance of the program. Using these tools the programer can then spend time thinking - for both MPI and OpenMP - how best to organise and access their data, making effective use of parallelism and concurrency.

Communication Sequential Processes provides a model for concurrent and parallel systems. This modal can be implemented by using the threading and communication primitives to provide the programer with a clear, runtime safe and structured view of their program. Implementing CSP can help the programer think and describe with grater clarity the structure of the program. For languages such as Python, this would allow a high level language to flexibly and clearly implement a program for high performance computers.
In order to investigate the usability of this modal and the abilities of Python as a language for high performance programming, this project will aim to take an implementation of CSP and investigate its use on a HPC system - ARCHER - comparing this by implementing a Graph500 solver. This program should aim to perform a simple version of Graph500 and discuss the results of bringing CSP with Python to HPC systems.

1.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP) [3], developed by Tony Hoare, is a language for describing concurrent systems using Process Calculi to mathematically design a parallel or concurrent system. These semantics that are used to define a system requires that process and there communication adhere to rules. One of the rules is that all processes are equal when executed and there is no order unless it has been imposed by the communication. The communication between these processes is described as a channel. These channels are considered to be blocking, and send message between the processes in the order they were placed on the channel. These strict rules allows programmers to accurately describe the behaviour of there system and prove mathematically the validity of the system. The justification for this can be considered similar to value of strongly typed programming languages. When use in programming systems these rules help prevent large concurrent or parallel programs from live or deadlocking during communication. This is similar to the reasons for a strongly typed programming languages; the rules of CSP can help prevent large concurrent or parallel programs from live or deadlocking during communication. This is very useful for safety critical programs but it can also apply to general programs, and these situations occur regularly when creating programs in MPI or other concurrent and parallel programs. In High Performance Computing should a program deadlock at a point in the simulation, this could mean having to rerun a very large simulation costing time and money. Programs designed using CSP can aim to have deadlocks and live-locks eliminated before the user runs there program. The mathematical principals behind CSP allows tools such as tools such as FDR [4] can validate a CSP program by checking that the designed system conforms to the rules, insuring the safety of the program.

While this project does not focus on the provability of a CSP system, it will test the feasibility of CSP for use on high performance computers by testing if the benefits of CSP can make a more robust and correct implementation of Graph500. CSP has been used in High Performance Computing in the past on the Meiko Compute Surface [5]. The Meiko made particular use of the Inmos Transputer [6, 7], which was programmed using OCCAM [8]. The Transputer process boards were arranged in a 2-dimensional array allowing each processor to be connected to some others. For CSP programs this means an individual process can reside on each processor and communicate in a similar topology on hardware as the system diagram used to design the program. OCCAM is the first implementation of CSP and many of the later implementations are influenced by the design choices made in the implementation of CSP for OCCAM. The internals of OCCAM can be found using the reference manual on the language [9], and its relation to other implementations can be found in Section 1.1.2.

It is useful to go through some of the imported features of CSP in more detail. These will be relevant to discussions on how to design systems, and there use in the development of Graph500. CSP programs fetters several basic building blocks that can be used to form the program.
**Processes**  Processes are units of work; When implemented these are often functions or a class object and can have a life time that is shorter that the totality of the program. In either case, they are single anonymous entities, Unlike ranks in MPI, where ranks 1 and 2 are copy of the same program that persist thought the lifetime of the program. While MPI process will operate on the one data set differently, a CSP process can complete a task or a group of similar operating tasks on the data set. Process can be have the same behaviour as MPI process, where each rank would be the equivalent of a process doing a particular task on the data set. While MPI process communicate via point to point messages at any time during the program, for a CSP process this is different. Each CSP process has a defined number of as inputs and outputs from the process, as apposed to communication from within the process based on a event.

**Channels**  CSP treats process as isolated objects; they must communicate through channels. Each process exists as its own entity they do not share the contents of there memory with other process outside of an explicit message communication. While the channel could be implemented as a shared memory construct, for the purposes of the model it is considered a message from one process to the other. The message between two process can be considered as an event. This event blocks the subscribed process has consumed the event. For system comprised of two process, each process can not progress until the event has been completed. Processes should have no restriction on the number of channels it can consume as long as they are treated in the same manner. Channel implementations can vary in how to achieve multiple senders and receivers but each multiple receive completes as soon as there is a matching send from one of the many senders. To manage more than one channel it can be useful to specify the construct of a *choice*. The process can then decide which channel to accept. Managing this *choice* is often implemented as an Alternative or Alt.

**Alts**  The *choice*, Alt, Alternative or Alternation, provides the programer an interface to pick between one or more channels on an action or by virtue of which channel is preferred over the other. This structure can allow other channels to be handled if there is data without blocking on the preferred channel. While this can make processes non-deterministic it does allow for large systems to be free of blocks as long as at least one of the channels has been resolved.

### 1.1.1 Building systems in CSP

The building blocks of a CSP, implemented in the preferred language of the programmer, can be used to build complete stable systems over an array of processors. Programmers can make use of the modular nature of their program to design the intended system before committing code to machine. This can be expressed in the formal notation or as a diagram. The latter will be used in this project to describe the design and implementation of Graph500.

Using Process calculus to describe a CSP program presents the mathematical view of the system. An example of such taken from Tony’s Hoare’s original paper can be seen in Figures 1.1 & 1.2.
\[ VMS = (\text{coin} \rightarrow (\text{choc} \rightarrow VMS)) \]

Figure 1.1: A vending machine that takes in a coin and returns a chocolate and returns to wait on a new coin.

This example shows a program that runs a vending machine. It excepts the input from the user of a coin. This is sent to the process and returns the desired item. The state at each stage of the process is described in the semantics of how the coin interacts with the vending machine. This is the mathematical representation of the system and by following the rules of this system, you can say for certain that the vending machine will return an item when given a coin. While this is a simple example, the formal description can expand to describe a more complex system. Figure 1.2 shows a more complicating vending machine.

\[ VMCT = \mu X \cdot \text{coin} \rightarrow (\text{choc} \rightarrow X \mid \text{toffee} \rightarrow X) \]

Figure 1.2: A vending machine that takes in a coin and returns a chocolate or toffee.

Depending on the preferences of a programmer, a visual representation of the vending machine can be more useful. Figure 2.4 shows a simple CSP program as a diagram. This is somewhat similar to some types of UML diagrams that show the relationships between the objects in the direction of the channels. These figures show the style that will be used throughout this project when designing components for Graph500. Both reading and writing ends will be labelled on a directional line joining two or more boxes, with each box labelled with the process name.

This system can also be drawn as a flow diagram. Each process is represented by a box and the directional lines show the reading and writing ends of the channels. The vending machine diagram has been overly expanded to show the full detail of the program. This can be implemented with fewer processes and channels. For bigger programs this method of designing CSP programs is often advised, as it gets the programmer to think about the communication and how the functions should interact with each other. The programmer can manually trace over the diagram in a similar manner to FDR or tracing programs that can evaluate the formal logic of the system. Time spent at this stage can help eliminate bugs that happen during development through communication problems. It does not protect against bugs that occur within the function, only in how processes communicate. Careful consideration should be taken when a process is consuming or generating more than one output, or when two processes have two or more channels between them. The programmer should think clearly in what order the channels should be consumed, or use an Alt to protect the channels.
1.1.2 Implementations

**OCCAM** OCCAM [9] was the first implementation used to implement CSP. It found commercial use on systems using the Inmos Transputer and has been ported to other systems.

**JCSP** JCSP [10] is one of the most progressed implementations of CSP for traditional programming languages. The library builds on the Java concurrency primitives and can be use with other JVM languages such as Groovy. JCSP make further implementation features from standard CSP by providing networked channels, and the ability to describe a process as mobile. This allows for the serialisation of processes across channels, providing mobile units of work.

**C++CSP** As of 2007 this library provides C++ with an implementation of CSP. [11] C++CSP2 provides the latest and upgraded implementation of CSP for C++. This library is influenced by JCSP and features much of the same breath of implementation, proving network channels and mobile processes.

**Go!** Go! [12] is a new language created by Rob Pike at Google. Its concurrence module takes direct inspirations from CSP and previous work by Rob Pike has featured CSP as an influence, e.g. large parts of the Plan9 operating system and the concurrency model of Limbo. While Go provides a channel object it not able to communicate with a process on a networked computer and behaves more like the occam channels. The syntax for defining channels and processes is markedly different from other implementations.

Any function can be used as a "goroutine" if the function is called with a "go" prefix. This behaves similar to a parallel statement, as the go verb tells the compiler to crate a lightweight thread for the function. Channels are called using a "<-" identifier which can be thought of as meaning "in to". The position of channel object denotes the object that is being written "in to". Channels in Go are synchronous and block when called, however a channel can be non-blocking by attaching a buffer. Messages are written to the attached buffer and the call returns immediately. Go is still a new language and there are currently very few examples of its use in scientific computing, however as its concurrency model is built-in as a first-class entity, it may prove to be useful.
It should be noted that Go’s implementation should be described as co-routines, and are context switched into use. To achieve parallelism, the compiler can be told about the maximum number of cores on the system, and schedule some of the routines to run on a other cores. However experience using one of the implementations carry over as the CSP modal is consistent throughout.

**PyCSP**  
PyCSP is the implementation of CSP for Python. It borrows from JCSP, and modifies the implementation accordingly to best suit python. More on PyCSP can be seen in Section 1.2.4
1.2 Python Language & Its use within HPC

1.2.1 Python Language

The Python language has found use within the scientific community for its ease of use in writing, brevity, extensive libraries and fast development time from idea to running code. While in use since 1991 and commonly thought of as a scripting language like Perl, it has been used more often to create fully featured programs and is often taught today as the first language for new students. For scientific purposes Python has found uses within the community.

The mainstream of programming languages for HPC scientific programming has been FORTRAN and C. Other languages have made an impact but most are languages that are statically typed and compiled. This gives the compiler a lot to work with and can make optimisations to the code before it is run. Python is not a compiled language and is dynamically typed, and this can present some challenges for its use. Python requires the use of an interpreter program to change the python program into byte code then into machine code. This may sound like Java but this interpreter does not make use of Just In Time compilation - like the Java VM - and there is an initial cost for compiling the code to byte code and then run. This does mean that successive runs of the code are faster, but both cases each line is interpreted as the program is run.

This does prevent the development cycle from having compile-time checks, where the program would be compiled and a list of warnings and errors can be seen before the program is run. This includes checking of types as this will happen when the lines are evaluated, however python enforces strict rules and, unlike Perl, an exception will be thrown should types be used inappropriately. Some of these problems can be mitigated by changing the way the program is developed.

The advantage of programming with an interpreter is that sections of code can be tested quickly, and a terminal can be open with the interpreter running to check and experiment on syntax. For example, the manipulation of lists or recursive functions can be checked quickly before being used in the design of the indented program. This can also be useful when trying out some of the libraries that are bundled with python or ones installed by the programmer.

Python is bundled with a rich library set that helps expand or provide the intended functionality. Some of these libraries are utilities that help the programmer debug, unit test or profile their code. These libraries are consumed with an import statement at the top of the program and contained within their own namespace. However, the use of Python in HPC has found that the import command, when consuming python library from the file-system, causes a major slow-down on large distributed-memory clusters. More information on this can be found in Section 1.2.3.

Python syntax is markedly less verbose than some languages such as Java, and enforces strict rules on whitespace to delimit the scope of block, and how types can interact with each other. The use of whitespace to delimit blocks of code can be considered frustrating, by programmers used to semicolons at the end of lines, and braces to surround code. This can be argued as a temporary discomfort and the benefit of enforcing this rule provides a degree of assurance, that looking at a page of python code with be of a certain level of clarity. This is especially useful for long standing code that may be maintained by several developers over a long period of time.

Python syntax is often remarked as being close to write english or pseudocode when typed.
An example of this high level syntax is the For loop. Shown in Figure 1.4 the python loop expresses the same as the others but in an expression closer to english. This can be beneficial for people who are beginning programming or people who find some other languages to be slower in composition.

```
# Python
for index in iterator_object:
    # do things with index

# C
for (int i = 0; i < N; i++) {
    # do things
}

# FORTRAN
DO i = 1,N
    ! do things
END DO

# Java Enhanced Loops for Arrays or implementers of Iterable
for (int x : array){
    # do things
}
```

Figure 1.4: For loop comparisons between Python, C, Fortran and Java

Similar to FORTRAN’s powerful array syntax, python has list comprehensions. These can be confusing to some programmers and can take time to create. An example is shown in Figure 1.5.

```
my_list = [x for x in list_of_A if x is G]
```

Figure 1.5: Python list comprehension

List comprehensions provide a powerful structure for iterating over a list to generate another list. This is often used for the construction of lists if conditions are met. There complexity arises when nesting comprehensions or providing complex conditions. They can be used for dictionary types as well.

Python’s syntax to define functions and classes is short, requiring only the definition that the block is a function or class, and what types it should take in or the inheriting class. Classes require the use of a special name `__init__`, which is the class constructor. Other special named under under names exist in python such as `__name__` to return the name of a function. Example syntax is shown in Figure 1.6.

Python’s type system is dynamic, and this can be called week typing as opposed to C or Java where the types are strictly enforced by the compiler. While the programmer does not need to tell Python what type a variable is, they do have to care about when and what they are
# Python Function

def func(var):
    # do lots of things
    return # stuff

# Python class
class Foo(superclass):
    def __init__(self, constructor_vars):
        # initialise constructor

    def Spam(self, num=1):
        for i in num:
            print "Spam!"

Figure 1.6: Declaration of Classes and Functions in Python

doing with it. This can result in some ambiguity when using types. In Java, to achieve a Dictionary(HashMap) requires a declaration of the HashMap class and what type the map will contain, which requires a verbose set of instructions, especially if the map should contain an Array or a second HashMap. In python, it only requires that the dictionary keys are a hashable type and unique. The value can then be any type and all that is required is inserting or constructing the dictionary with the intended type. While each is achieving the same thing, the advantage of not worrying about the types - to the extent of declarations - allows the programmer to quickly get work done and spend time thinking about what should be done. The dictionary can now contain a varied set of data such as a nested dictionary where some elements might be lists and some integers or strings.

Python programs can benefit - when used appropriately - from mixing functional, procedural and object oriented design patterns without compromising the structure of the program adversely. It should be noted that the philosophy of python is to find one good way of doing something and do that often. An example of the added functional style is the use of lambdas. This allows for a named function declared using the syntax shown in Figure 1.7.

```
power_2 = lambda n: n**2

print power_2(8) # result -> 64
```

Figure 1.7: Python’s Lambda syntax

This can be used where it is required in the program, e.g. for an explicit single use function within a section of code, or where naming a permanent function in the scope above is unnecessary.

The stack and heap are automatically managed by pythons garbage collection, though objects can be tagged for deletion using the ‘del’ command.

The features of python that make it comfortable and easy to program come at a cost. When programming in C it does not manage memory for the programmer, and care must be taken when
packing structures the size of arrays, and the iteration through memory. Python performs by
default worse when doing these operations such as inserting into lists or deep copy of memory.
Python’s performance can be improved by binding python with C directly.

Libraries such as Cython [13] and ctypes [14] allow the programmer to move performance
critical components to C without diverging from python’s philosophy greatly.

**Decorators** Python allows for a function to wrap other functions, which can be used to modify
how the function operates. Functions can be defined within others, and inherit their scope.
In current versions of python, this functionality is provided by the ‘@’ symbol, as shown in
Figure 1.8.

```python
@Spam
def breakfast(word):
    return word
```

Figure 1.8: Python decorator example

### 1.2.2 Parallelism and Concurrency

Python provides modules to create threads or processes to allow parallelism and concurrency.
The main modules are `threading` and `multiprocessing`. There is a lower level API to threads and
greenlets, which implement a very lightweight thread [15]. As the PyCSP library makes use of
these, below is a brief overview of each with an awareness for some of the restrictions placed
by each module.

**Threads** Threads are provided in the `threading` module and used in PyCSP as `@process` dec-
oration. Threads in python are considered by some to have poor performance [16][p216]. How-
ever, this is mostly in cases where the program is threading a CPU bound section of code.
Python threads tend to be used in cases where I/O tasks are required or where the overhead is
less than the single-threaded performance for the task. The main issue that is cited is the Global
Interpreter Lock (GIL). This lock enforces only one thread currently executing byte code on the
interpreter at a given time.

**GIL** The Global Interpreter Lock exists to prevent two threads clashing on the interpreter,
making sure only one is running code on the interpreter at a time. It also makes the implemen-
tation for managing and scheduling threads easier to implement in the Python interpreter. The
GIL is released when a thread enters a section of byte code for I/O or an external call that can
use its own native threading. The other condition is if a thread has owned the lock for a certain amount of time. This can be a count - the number of machine instructions progressed - of a 100 and when a thread has reached this condition the lock will be given to the next waiting thread.

For I/O bound tasks, this implementation does not cause a problem; non-scientific code is more likely to read/write files or make use of devices such as sockets. A thread blocked for an I/O call can release the GIL and another thread can lock the interpreter for processing. In some situations there can be contention on the lock when the previous thread wakes from I/O resulting in a reduction in performance. The programmer should consider what work they wish to do and choose the most appropriate tool. If the work is mostly CPU bound the preferred tool is to use the multiprocessing module.

**MultiProcess** The `multiprocess` module provides the same functionality as the threading module but with some changes to the underlying operation. Instead of creating a thread, the module will make use of the sub-process call to create a new OS process. There are restrictions on the interaction of variables with this new process, as it requires a message passing system and cannot share variables between processes. The benefit of creating a second process from the interpreter is that it is not bound by the GIL. This gives python the ability to 'thread' CPU bound code and achieve the required increase in performance. PyCSP makes use of this module through the use of `@multiprocess` decorator.

### 1.2.3 Python Limitations in HPC

There are some limitations found within Python that can impede its use for High Performance Computing. However, there are efforts to mitigate this allowing programming to use the best quality in python without sacrificing performance.

Algorithm performance can be improved by writing portions of the code in C or Fortran. Often the programmer can use a currently existing library in C or Fortran and consume them within python. The two most prominent libraries for scientific computing are Numpy and SciPy.

**Numpy** Numpy [17] features a comprehensive package for mathematics with significant use of arrays and matrices. This allows for large arrays in N dimensions to be stored and indexed, with methods for matrix-matrix multiplication, BLAS [18, 19] functions and integration with python’s built in data structures.

**SciPy** SciPy [20] offers further scientific modules such as LAPACK [21], Fourier Transform, Sparse Matrices and signal processing. SciPy can make use of numpy arrays, allowing this library to make use of sparse matrices.

**Mpi4Py** Python can make use of MPI though the Mpi4Py [22] module. This uses the C bindings to provide MPI2.2 support to python. Moving from the C to python libraries can be slightly confusing. `MPI_Init` happens at the point when the module is imported at the top of the program `from mpi4py import MPI`. Once the MPI namespace import has initiated MPI, the communicator can be retrieved as `MPI.COMM_WORLD`. All subsequent interaction with MPI
functions are a method of the communicator object. The library defines two copies of each method. An example of such is shown in Figure 1.9.

```python
comm = MPI.COMM_WORLD
comm.ssend(obj, dest=1, tag=0)
comm.Ssend(buffer, dest=1, tag=0)
```

Figure 1.9: An example of a send in Mpi4Py with each of the two types of sends.

The difference between the two methods is that the lower case method takes in any python type object, whereas the upper case method only takes a buffer object. The buffer object is a distinct type in python, and its use is preferred when sending Numpy arrays. Numpy arrays can be referred to by a pointer. The buffer object becomes this pointer to the memory allocated to the numpy array. This also helps improve performance as the type of the array can be specified. This saves time on introspection and the time taken to serialise a python object before sending. Other differences are often that the C version would update an variable from a pointer, the python version will have the value returned from the method.

**DLFM**  
Pythons use in High Performance Computing has revealed that on large distributed memory supercomputers, importing modules adds a significant overhead when using large numbers of python instances.

When python programs are distributed over the compute nodes, each program will call the same set of import lines at the top of the program. This will mean a large number of requests to load the library from the file-system, resulting in a bottleneck. EPCC has researched one possible way round this by implementing a program named DLFM [23], which distributes the dynamic lib rays(dlcache.dat) and the python file models (fmcache.dat). In a a white paper presenting DLFM [23] and its scalability using GPAW [24] as an example, resulting in a marked improvement in the import time. DLFM takes one of the cached modules and distributes it amongst the nodes. The nodes load the library from this cached file that was sent to them. This reduces the number of python programs requesting modules from the file-system and speeds up the time taken to load all the modules. The white paper reports that a drastic improvement over the standard import mechanism was achieved.

**1.2.4 PyCSP**

PyCSP was first released in 2007 and has since made progress to its current implementation. While boring heavily from JCSP, some changes have been made to better appeal to the user base and better fit the python language.

To implement processes PyCSP defines:

- `@process` This defines a standard process using the threading library
- `@multiprocess` This is the same as process however it makes use of the multiprocessing module
Other process types included in PyCSP

@sshprocess defining process on a particular host

@clusterprocess a process that will be part of a cluster using the PBS scheduler

The main types used are the process and multiprocess processes. These are implemented as
decorators and are placed in the line above the declaration of a function.

Channels are objects using the Channel constructor, and each have a read method - short-hand
symbol a plus (+) - and a write method - short-hand symbol a minus (-). PyCSP implements
Channels as all-to-all. Channels are not typed and there are no special cases for all-to-one or
other relationships found in JCSP. This allows PyCSP channels to have as many readers and
writers as required by the programmer. Channel objects do not mandate the type of data that
can be sent along them, other than the objects must be serialisable.

PyCSP implements the choice in Alternation.py. Alternation provides Fair, Priority and Default
selection functions as a wrapper for the main Alternation implementation allowing the pro-
gramer to modify the behaviour of the choice. When channels are consumed by the alternation,
the programmer can give preference or load balance between which channels are consumed.
The Alternation - Alts for brevity - are needed since CSP channel is a blocking send and re-
ceive.

The Alt consumes channels by using a Guard object and each channel in the Alt is a list of these
Guards. The Alt will iterate over the Guards to test if a channel event has occurred and return the
first Guard that has a completed event. If a Guard has not completed, the Alt will proceed to the
next guard in its list. The Alt can take 4 types of guards, InputGuard, OutputGuard, SkipGuard
and TimeoutGuard. The InputGuard wraps a channel_read end and this has been updated for
MPI channels. OutputGuards wrap the channel_write end provide a similar functionality to the
Input Guard. The last two Guards are special with Skip allowing the alternative to define a
default action, and Timeout to provide a timed use of the guard before terminating the Alt.

Channels can be terminated by ‘poisoning’ or ‘retiring’ the channel when it should stop being
used. This frees up resources and can be used to cleanly - in the case of retiring - stop all
communication before the program terminates.

To create processes, the ‘Parallel’ method is used and takes a list of process and there channels.
It is also possible to add a multiply symbol (*) followed by the number of process required and
the parallel will create that many copies of the process object.

PyCSP so far seems to have been tested on commodity clusters but shows it can scale well and
successfully implement algorithms for scientific computing. For example, an implementation
of k-nearest neighbourhood search and a token ring test [25] shows that, with modifications,
PyCSP can scale to 512 CPU cores - each running a process - with the same number of channels.
This shows that it should be possible to run PyCSP on ARCHER at this level, possibly with
some modifications.
1.3 Graph500

The Graph500 [2] benchmark was developed as an data-intensive benchmark that provides a performance result that was more than just the floating-point performance of a given computer. This should lead to a more accurate metric for performance when estimating every day physics simulation applications in use by researchers. While the main focus of this project is not Graph500, this benchmark is used as an example application in this project. The Graph500 specification and the requirements for implementation are outlined in this section. The project’s implementation of Graph500 will be tested against the sample MPI implementations supplied by the Graph500 Committee, but this will not be a test of ARCHER’s performance in completing the benchmark. This will be used to assess whether Python/CSP implementation has sufficient coverage to satisfy the requirements of a valid implementation.

1.3.1 Specification

A valid Graph500 benchmark implementations must be comprised of two kernels. The first kernel constructs an undirected graph from an edge-list. This kernel is timed as well as the second kernel. The second kernel performs a breadth first search (BFS) on the graph. The graph must be searched 64 times with each iteration selecting a randomly generated starting point. Once an iteration is complete the search is validated and then the next iteration can begin. The output should display the times for each stage and the two kernels - the first times for graph generation and constructions, the latter kernel times for BFS completion and validation. The end of the output displays statistics and settings used for the instance of the benchmark.

<table>
<thead>
<tr>
<th>Problem class</th>
<th>Scale</th>
<th>Edge factor</th>
<th>Approx. storage size in TB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy (level 10)</td>
<td>26</td>
<td>16</td>
<td>0.0172</td>
</tr>
<tr>
<td>Mini (level 11)</td>
<td>29</td>
<td>16</td>
<td>0.1374</td>
</tr>
<tr>
<td>Small (level 12)</td>
<td>32</td>
<td>16</td>
<td>1.0995</td>
</tr>
<tr>
<td>Medium (level 13)</td>
<td>36</td>
<td>16</td>
<td>17.5922</td>
</tr>
<tr>
<td>Large (level 14)</td>
<td>39</td>
<td>16</td>
<td>140.7375</td>
</tr>
<tr>
<td>Huge (level 15)</td>
<td>42</td>
<td>16</td>
<td>1125.8999</td>
</tr>
</tbody>
</table>

Table 1.1: Graph500 Problem Class definitions [2]

Graph500 can be run with different classes as defined by the scale of the number of vertices in the system - the edge factor usually remains at 16. This defines the number of connections a vertex has. The classes increase in size and a full table can be seen in Table 1.1

1.3.2 Graph Generation

The graph is constructed as a list of tuples were the first element stores the start point vertex and the second the end point vertex. This edge list has some rules that must be observed. The list cannot indicate locality as this is the job of the graph creator kernel. The edge list will be modified into a data structure that will be distributed. There is no restriction on the type of data structure; this could be in the form of a sparse matrix, linked-list or hash-table. The decision on the best structure is left to the implementer.
The Graph500 Committee provides a sample code in GNU Octave [26] - a matlab like program. Figure 1.10 shows a a simpler version of the generator that is provided in the sample code.

```matlab
function ij = kronecker_generator (SCALE, edgefactor)
% Generate an edgelist according to the Graph500 parameters. In this sample, the edge list is returned in an array with two rows, where StartVertex is first row and EndVertex is the second. The vertex labels start at zero.

% Example, creating a sparse matrix for viewing:
ij = kronecker_generator (10, 16);
G = sparse ((ij(1,:))+1, ij(2,:)+1, ones (1, size (ij, 2))); spy (G);

% Set number of vertices.
N = 2^SCALE;

% Set number of edges.
M = edgefactor * N;

% Set initiator probabilities.
[A, B, C] = deal (0.57, 0.19, 0.19);

% Create index arrays.
ij = ones (2, M);

% Loop over each order of bit.
ab = A + B;
c_norm = C/(1 - (A + B));
a_norm = A/(A + B);

for ib = 1:SCALE,
% Compare with probabilities and set bits of indices.
ii_bit = rand (1, M) > ab;
jj_bit = rand (1, M) > ( c_norm * ii_bit + a_norm * not (ii_bit) );
ij = ij + 2^(ib-1) * [ii_bit; jj_bit];
end

% Permute vertex labels
p = randperm (N);
ij = p(ij);

% Permute the edge list
p = randperm (M);
ij = ij(:, p);

% Adjust to zero-based labels.
ij = ij - 1;
end
```

Figure 1.10: Sample GNU Octave graph generator supplied by Graph500

Both the sample and the octave code in generate a list of edges. The sample code will take this
edge list and use an adjunct matrix. This is a symmetric space matrix that is used to store the
graph. These data structures, once distributed, cannot be modified by any of the subsequent
parts of the program. After the creation and distribution of the graph, 64 search keys are to be
created. These are the points in the graph that become the root node in the Breadth First Search.

1.3.3 Breadth First Search

Graph500 provides a repository of reference code. This contains examples of sequential, OpenMP,
MPI and XMT versions of code that implement Graph500. In this project, the serial and MPI
sample code will be used during the development with the MPI version used as the sample code
to compare the python-csp version. The reference code also provides a separate example of the
graph generator in C. The MPI solver is implemented in bfs_simple.c with other version that ex-
plot the different features of MPI such as single sided communications. The simple one is used
to help simplify the requirements of the python CSP version and in the functional comparison
between the two versions.

1.3.4 Implementations

Graph500 provides a repository of reference code. This contains examples of sequential, OpenMP,
MPI and XMT versions of code that implement Graph500. In this project, the serial and MPI
sample code will be used during the development with the MPI version used as the sample code
to compare the python-csp version. The reference code also provides a separate example of the
graph generator in C.
Chapter 2

Python/CSP implementation

In attempting to implement Graph500 using PyCSP on ARCHER it required setting up a copy of the library using virtualenv. PyCSP is not provided on ARCHER by default but can be installed from the Python Package Index using the pip command. virtualenv is used on a directory to create a local instance of python and a local linked set of library. pip is run from within this environment to install PyCSP. Once set up a PBS job can be run sourcing the local environment and running a PyCSP program.

2.1 PyCSP Process Creation

Process in PyCSP are created by listing the required processes in the Parallel() function. This however does not have any knowledge of architecture of the machine and can’t make any intelligent placement of process. Since all processes are instances of threading or multiprocess and are local to the interpreter any program that should scale beyond one node would need a copy of the interpreter.

For ARCHER programs are placed on the compute nodes using the aprun command. This is part of ALPS [27] which manages and allocates process across the compute nodes. Using aprun now allows the scheduler to create N of copies of the PyCSP program across ARCHER. An example of a PBS script can be seen in Figure 2.1. However this has just created clones of the same program. Each PyCSP defines the same parallel and will execute the process without knowledge of each other. To amend this, PyCSP must know which process are assigned to each node, allowing the creation of a Parallel that is run across nodes with the ability to run a second parallel within the distributed PyCSP processes.

2.1.1 Distributed PyCSP Processes

The Parallel in the current implementation is a wrapper around a function that takes in a list of processes. Theses are then iterated over and the start method is called. This is an inherited method but the processes being decorated by the threading and multiprocess modules. This can be seen in Figure 2.2.

The Parallel variable plist holds the processes that were declared by the programmer. This function - _parallel - will need to be amended to allow processes to be distributed over ARCHER.
During development there was no visible way to implement this using an API from cray to achieve similar to the distribution of MPI programmes. To solve this problem the _parallel function would be modified to use MPI via the mpi4py module.

Each copy of the PyCSP program should initialise MPI, and obtain a rank, and the size of the pool from MPI_COMM_WORLD. Now each copy of the program has this information from MPI they can use this to run a process from the list.

Figure 2.3 shows the processes passed to the Parallel function and the modification to the private parallel function now only being called with one of the processes. Since plist can be indexed by rank number, the first process in the list is assigned rank 0 and so on to the size of the communicator. This now allows PyCSP to run processes over ARCHER.

Other solutions to this discussed; serialising and sending process from rank 0 to other processes. This would allow more than one process to be issued per rank but the scaling was deemed worse than having each copy pick process to run. This would be useful work for mobile process. Some CSP implementations allow for process to be sent via channels using them as a mobile task. This could also be used in situations where CSP processes are dynamically instanced based on the required decomposition of work. Should a node have more work than required a mobile process can sent to redistribute work and move to a quieter node. One other prospective solution to distributed process on ARCHER was the use of the @clusterprocess decorator. This process is able to read the PBS node file that lists all the compute nodes hostnames. As ARCHER does not use the PBS system in full - for standard clusters hydra is used where ALPS is used on ARCHER - and the node file is not available to users.

Now that process can be distributed there is the issue of communicating between processes.
```python
def Parallel(*plist, **kwargs):
    return _parallel(plist, True, kwargs)

def _parallel(plist, block, kwargs):
    processes = []
    for p in plist:
        if type(p) == list:
            for q in p:
                processes.append(q)
        else:
            processes.append(p)

    # Update process parameters
    if kwargs:
        for p in processes:
            p.update(**kwargs)

    # Start processes
    for p in processes:
        p.start()

    if block:
        # Blocking behaviour
        return_values = []
        for p in processes:
            return_values.append(p.join_report())
        return return_values
    else:
        # Spawn
        p, _ = getThreadAndName()
        p.spawned.extend(processes)
```

Figure 2.2: PyCSP Parallel functions for process instantiation. Docstrings have been removed for figure clarity

```python
# outside the parallel function
Parallel(process_1(chan+, chan2-),
         process_2(chan-, chan3+), process_3(chan2+, chan3-))

# inside the parallel function
return _parallel(plist[rank], True, kwargs)
```

Figure 2.3: Modification to provide each MPI Rank with a process to run

### 2.2 PyCSP Inter-process Communication

When designing PyCSP programs - and CSP programs in general - they may start with a connected diagram of the processes. Figure 2.4 shows an example diagram of 3 connected pro-
cesses. For thoraces process to communicate between nodes some level of network channel is required.

PyCSP has the ability to create networked channels by passing the address of the terminating end of the channel in the channel constructor. A simple example program is shown in Figure 2.5 and 2.6. The Figures show one program defining a 'source' where data will be written and another program defining a 'sink' where data will be received. The 'source' program specifies the network address in line 8 of the sink program. This is implemented using TCP/IP sockets and allows the programmer to communicate between distributed process.

```python
import pycsp.parallel as csp

@csp.process
def sink(channel_in):
    print(channel_in())

par = csp.Par()
chan = csp.Channel("A")
par.Parallel(-chan)
csp.shutdown()
```

This implementation of networked channels is not able to work on ARCHER’s compute nodes. ARCHER features the Cray XC30's Aries interconnect, and this is implemented using Infini-band. The compute nodes use this as their own network secret from any front end node the programmer launched the PyCSP job from. Since the PyCSP channel assumes that the programmer has prior knowledge of which nodes are hosting each subscribing processes, on ARCHER the programmer is not able to acquire this information before hand. Not only will two jobs be unlikely scheduled in the same place, there is not command to proper reserve nodes and no list of hosts from the programmers view of ARCHER.

MPI can be used in place of the current implementation of network channels as it has solved some discover problems. MPI knows the location of the node using a particular rank number and can correctly send messages between processes. Since processes are now being distributed with one per rank, the communications relationship shown in Figure 2.4 can be expressed.
```python
import pycsp.parallel as csp

csp.

def source(channel_out):
    print(channel_out("Spam Spam Spam"))

par = csp.Par()
chan = csp.Channel("A", connect=(192.168.1.1, 5678))
par.Parallel(+chan)
csp.shutdown()
```

Figure 2.6: Network channel example showing the writing end `source`

PyCSP channels are designed to have one `channel` class and a subclass of `channel_end`. This is then further sub-classed for each type of end - read or write. This can be replicated for the new `ChannelMPI` class. Each channel end is amended to implement a private method of the channel object. `ChannelMPIEndWrite` will call the method for posting an MPI_Ssend, and `ChannelMPIEndRead` will post an MPI_Recv. In order to send each channel must know the rank sending and source of the sent message. The best place to add this functionality is in the `Parallel` function.

```python
if self.MPI:
    comm = MPI.COMM_WORLD
    world = comm.Get_size()
    if world >= 1:
        rank = comm.Get_rank()
        if plist[rank].args:
            channel_ends = [[end for end in proc.args] for proc in plist]
            list_ends = [end for ends in channel_ends for end in ends]
            my_ends = channel_ends[rank]
            channel_index = [channel_ends.index(ends) for ends in channel_ends]
            channel_names = list(set(chan.channel.name for chan in list_ends))
            channel_graph = {}
            for my_end in my_ends:
                for end in list_ends:
                    if my_end.channel.name == end.channel.name
                        and end.end_type != my_end.end_type:
                            end_rank = [channel_ends.index(ends)
                                        for ends in channel_ends for e in ends if e is end]
                            channel_graph[my.end.channel.name] = [my_end,
                                                       rank, end, end_rank[0]]
                            end.channel.set_graph(channel_graph)
                            end.channel.set_comm(comm)
                            end.channel.set_tag(channel_names.index(end.channel.name))
            return self._parallel_mpi(plist[rank], True, kwargs)
```

Figure 2.7: Distributed channel management for Distributed Processes

Figure 2.7 shows the code required to set up networked channels. Since each node is now running this code they can each enumerate the channels from their view of the system. All
processes have knowledge of their own channels and the other processes, and can set up a data structure - *channel_graph* - to store each channel and the endpoints of that channel. Line 13-19 loops over the channel data structures to find the opposite end for each of the ends that the current process owns. These ends are entered into the *channel_graph* with the rank of the process. Within this loop the rank is discovered by the index of the channel in the process list. Lines 21 to 23 update the channel object with the *channel_graph*, a copy of the MPI_COMM object and the tag. Each channel’s tag is the index value of the channels name in an ordered list. To match the rules of CSP, the MPI channels use blocking calls to implement the blocking nature of the sockets implementation.

With a working implementation of MPI channels, the final view of the PyCSP program on ARCHER can be seen in Figure 2.8. Each process described in the original diagram and placed in the parallel, is assigned a node and can now communicate via a point to point channel between nodes.

![Figure 2.8: Physical view of distributed PyCSP processes with networked channels](image)

### 2.2.1 Alternative Solutions and Considerations

When starting processes using MPI, *aprun* will need to have the correct number of processes that will be instantiated within the parallel. Misinforming *aprun* will result in an error from the parallel that the list index was out of range for a given rank, or an error during runtime as a process has not been allocated a rank.

Implementing channels with point-to-point MPI functions does restrict any improvements that can be gained by using one of the MPI collective communications. If it is known that the channel is sending to more than one process, it would perform better to use MPI_Bcast, MPI_Scatter or MPI_Alltoall. While this was considered, channels implementing collectives were considered too complex, requiring their own sub-communicator to correctly send and receive the data to the correct subscribing processes. This would need to be investigated in further work.
This implementation requires further work to provide better support for multiple channels. Some of this work has been detailed in Section 2.3.

### 2.2.2 Syntax for modified Parallel and modified Channels

An effort was made to try and minimise the changes that implementing processes and channels for ARCHER would have on the syntax of PyCSP. One syntax change does break backwards compatibility but provides a clearer and maintainable inference when using a parallel. The parallel function was originally a function within the process class. However, Parallel is not the only function for creating process and there is Spawn and Sequence. These have been moved into their own class. The Parallel function is now a member of the Par() class. It is now clearer when creating a parallel and which Parallel is using MPI or not. This is handled using a option in the Par() constructor. Creating a Parallel using MPI, the programmer must set Par(MPI=True) when instantiating the Par object.

MPI channels are used by instancing ChannelMPI instead of Channel. The postfix of MPI make it easier to switch between channel implementations and a clear distinction between which channel implementation is in use.

### 2.2.3 Alts! (Alternatives) and Guards

On completing process distribution and network channels, a consideration of how to implement graph500 discovered that more work on the infrastructure was required, namely the PyCSP feature of Guards and the Alternative.

To allow a process to better consume more than one MPI channel the Alt implementation was updated to add support for MPI defined channels. So as not to change the functionality of the Alt such that it deviates from the correct behaviour and the additions break any insisting functionality, the Alt now has some added checking for the type of channel. In the new implementation there has been no change in the implementation of [Alt | Pri | Far]Select functions and the changes have been in the Alternation class, and CSP MPI Channels.

When the Alt is used, it first calls a method execute with the list of guards passed. This is internally consumed by the _choice() method. For each guard the channel is then inspected and the channels _post_read() or _post_send() method called. This will create the equivalent of a non-blocking communication, wait for the channel to be acknowledged and read the message buffer. This behaviour is translated into the MPI equivalent for networked channels.

Within the Alt the _choice() method is amended to check if the channel in the guard is an MPI channel. With the MPI flag set the MPI channels _post_read() method is called. Any other checks made by the alt are ignored if the MPI flag has been set. The alt now needs to test for the completion of the MPI communication. Figure 2.9 shows this change to the _choice() method.

Some of the existing infrastructure has be to used such as line 14 where the process thread running the CSP process needs to know that the channel completed. This is due to a test later in the Alternative code that is reset after the completion of the Alt.

The implementation uses MPI_Waitany by collecting the non blocking receive object from all the channels in the Alt and waiting until one has been resolved. This matches the existing
if chan.type == "MPI":
    chan_type = "MPI"
    chan._post_read(proc)

if chan_type is "MPI":
    request = []
    status = Status()
    for chan in reqs.keys(): # move that up to post and discard result of active
        if chan.active:
            request.append(chan.channel.irecv_result)
        else:
            chan._post_read(proc)
    wait_id, msg = Request.waitany(request, status)
    proc.state = SUCCESS
    for chan in self.guards:
        if chan[0].channel.tag == status.tag:
            proc.result_ch = chan[0].channel.name
            chan[0].active = False

Figure 2.9: Amendments to Alternative.py to allow for MPI channels.

The mpi_channels have been amended to implement non-blocking communication. Figure 2.10 shows the additions to the channel end.

Figure 2.10: Modifications made to mpi_channels

The channel stores the result of the non-blocking communication as a member attribute that is read by the alternative. The channel end will post a non-blocking receive only if the channel flag active has not been set, which avoids dangling communications.

This now allows all types of Alts to be used on MPI channels, allowing a node to read and manage multiple distributed processes. MPI Channels have also been tested to check that the action on a guard can complete, allowing for the use of a function decorated as @choice.
2.3 Multiple Readers and Writers of networked channels

During the project it was realised that multiple readers and writers would be required to expand beyond one node, however during discussions it was not clear in what way they should be implemented, and this lead to it being moved out of scope for this project.

Part of the issue with this is using MPI. For multiple channels using sockets, while there will still be an atomic operation on the socket it can address one to any and any to one channels without major changes to the underling operation. For an MPI based channel there is consideration into the optimal implementation. Using a collective communication that will be optimised for bulk sending of buffers to multiple subscribers, or a rudimentary implementation using multiple point to point sends.

![PyCSP Graph Solver graph with MPI rank numbers](image)

Figure 2.11: PyCSP Graph Solver graph with MPI rank numbers

Figure 2.11 shows the implemented of the PyCSP breadth first search diagram with the red numbers indicating the assigned rank for each process. It can be seen from this diagram that the Graph distributor needs a one to any channel, the workers requiring a any to one and the queue sender to have one to any. To try to implement this relationship in PyCSP using MPI, it will require changing the parallel method and the mpi_channel code.

During the project, a restricted time period of 2-3 days was allocated to implement a basic multiple point to point send and evaluate what would required for a full implementation. The design was to count the number of ends that the channel had and issue the corresponding number of sends and receives. This is not an optimal implementation as it requires a loop over point to point channels, negating any optimisation that can be found in an MPI collective communication and requiring a call for every communication required.

The necessary changes to the parallel.py code was to record the correct number of the opposite ends of the channel, and their rank. Figure 2.12 shows the updated end discovery with the
for my_end in my_ends:
    end_count = 0
    for end in list_ends:
        if my_end.channel.name == end.channel.name and end.end_type != my_end.end_type:
            # end_rank = [channel_ends.index(ends) for ends in channel_ends for e in ends if e is end]
            end_rank = self.list_duplicates(channel_ends,[ends for ends in channel_ends for e in ends if e is end][0])
            channel_graph[my_end.channel.name] = [my_end, rank, end, end_rank]

Figure 2.12: Updated channel end discovery

original code commented out. Originally line 5 would unpack the list of channel ends to create a flat list, and search for the matching index of the current end. While this works fine for one end and returns the rank of the opposite end of the current ranks end, it does not account for duplicate ends. If there are two channel_end_reads of the same channel object only the first one will be correctly indexed.

The new implementation returns a list of ranks for each matching channel end. This now allows each channel to know who the sender or receiver is of each channel end. This still remains a point to point message only now if there are two senders, two matching receives for each send will be posted.

The required code changes to mpi_channel are shown in Figure 2.13

#### Figure 2.13: Changes made to mpi_channel send. Similar changes are made for recv

This implementation does work but there is an outstanding communication or a failure to terminate the running process and the program completes its work but remains running. However, it does show that this is possible and more work is required. Section 5.1 details some of the considerations for further work.
2.4 Graph Generation

The Graph500 specification provided a sample graph generator showing previously in Figure 1.10. A serial version of the graph generator was built to understand how the graph structure is built for the development of the python breadth first search.

```python
import numpy as np
import numpy.ma as ma
import math
import random

def graph_generator(scale, edgefactor=16):
    N = int(math.pow(2, scale))
    M = edgefactor * N
    A = 0.57
    B = 0.19
    C = 0.19

    ij = np.ones((2,M))

    ab = A+B
    c_norm = C/(1-ab)
    a_norm = A/ab

    for ib in range(scale):
        ii_rand = np.random.rand(1,M)
        ii_bit = ma.filled(ma.masked_where(ii_rand > ab, ii_rand),0)

        jj_rand = np.random.rand(1,M)
        ii_not = np.logical_not(ii_bit).astype(int)
        jj_bit = ma.filled(ma.masked_where(jj_rand > (c_norm * ii_bit +
                                                    a_norm * ii_not), jj_rand),0)

        ii jj = np.concatenate((ii_bit, jj_bit))
        ij = ij + math.pow((ib-1), 2) * ii jj

        #np.random.shuffle(ij)

    ij = ij.astype(int)-1
    ij = ij.reshape(M,2)
```

Figure 2.14: Python serial version of the sample graph generation

Figure 2.14 implements the Octave sample using Numpy. It uses the parameters supplied to create a power of 2 times the scale input value and multiples this by the edge factor of 16. This creates a large number which defines the number of edges. An empty array is created and then populated storing two numbers for the calculated number of edges. This program returns this two dimensional array as the edge list and this is translated into the sparse matrix that represents the graph.
This program was checked against the Octave supplied program and the C version to confirm the output from the python version. While both worked, the C sample code implemented a seed which allowed the generated the same graph for each scale factor. The python generated graph was run with the C versions Breadth first search worker and this validated the output. While the python version was generating correct graphs, it was decided that the main priority was to implement the breadth first search. The C version producing stable and reproducible graphs was a candidate to use over the python implemented version as this prevents subtle changes in the graphs altering the results.

2.5 Breadth First Search

2.5.1 Design

To design the PythonCSP version of the breadth first search the first task was tracing and understanding the approach of the C implementation.

Figure 2.15 shows a section of the Vampir trace for the graph500 sample program. The program was run with a small graph and a small number of nodes to make the trace easier to observe. It can be noted that the program is dominated by communication; there are very few areas of calculation within the program - denoted by the green areas in each process bar. The red zones indicate time the program spent on communication, with the black lines showing the direction of the MPI communication and the MPI call. MPI_Isends dominate the communication between ranks as they exchange vertices, with a reduction at the end to make sure each rank is on the same level in the graph before proceeding to the next level. Noting this interaction and reading the code can help visualise the interaction between the ranks when reading the sample code.

The program defines bfs_simple.c which implements the breadth first search, with main.c having created the adjunct matrix and distributed the graph amongst the ranks. The code has comments indicating that bfs_simple.c implements a 1 dimensional level synchronous search.

For this project, the Python-CSP version implements the same algorithm, as closely as possible, and traverses the same graph as the C generated version.

The first step in implementing, was to find a pseudo-code implementation of the algorithm. The C version contains a large body of code to implement the algorithm in a C style and this should not be copied across, as the python implementation should be written to make best use of the high level features in python. Some of the work required in the C version will be implemented in functions provided within python or through a scientific library.

Figure 2.16 shows a pseudo-code implementation presented at the Supercomputing 2005 Conference [28]. This implementation creates a data structure to store the level of each vertices as the program depends down each level. Each processor is given a slice of the adjunct matrix and stores a local set of vertices it owns. The program will find the neighbours of each vertex - some of which may be on other processors - and then swap vertices with each of the other processors. By creating a union of the incoming neighbours and the local neighbours, each processor determines the vertices for that level, and can record the visitation of the vertex at that level. This implementation is not the best performing; there are other implementations using a combination of top-down and bottom-up searching as well as implementations decomposing.
Figure 2.15: Vampier Trace of graph500 sample C/MPI
Algorithm 1 Distributed Breadth-First Expansion with 1D Partitioning

1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{ where } v_s \text{ is a source} \\ \infty, & \text{otherwise} \end{cases}$

2: for $l = 0$ to $\infty$ do
3: $F \leftarrow \{ v \mid L_{v_s}(v) = l \}$, the set of local vertices with level $l$
4: if $F = \emptyset$ for all processors then
5: Terminate main loop
6: end if
7: $N \leftarrow \{ \text{neighbors of vertices in } F \text{ (not necessarily local)} \}$
8: for all processors $q$ do
9: $N_q \leftarrow \{ \text{vertices in } N \text{ owned by processor } q \}$
10: Send $N_q$ to processor $q$
11: Receive $\hat{N}_q$ from processor $q$
12: end for
13: $N \leftarrow \bigcup_q \hat{N}_q$ (The $\hat{N}_q$ may overlap)
14: for $v \in N$ and $L_{v_s}(v) = \infty$ do
15: $L_{v_s}(v) \leftarrow l + 1$
16: end for
17: end for

Figure 2.16: Distributed Breadth-First Expansion with 1D Partitioning from [28]

the graph matrix in 2D. These implementations provide a greater speed-up in the total number of edges visited and the scalability of large graphs over thousands of nodes. [29]

The communication between processors can be seen in line 10 and 11 of Figure 2.16 and show a point-to-point relationship with the neighbours in the graph. The C version implements this by making a note of all processors that are requesting vertices and matching each request until all have been satisfied and there are no more vertices to send and each vertex in the frontier has been searched. For the PyCSP version, since all channels are blocking it is not possible to connect two processes directly, as both would block waiting to receive the other’s send. This could be alleviated with a buffered channel, but at some point the channel would block if the buffer space was full. The safest way is to add processes to coordinate messages between processes working on the breadth first search.

Figure 2.17 shows the process and communication direction for the implementation of the breadth first search. The graph is decomposed in the graph distributor from reading in the edge list from file. Each worker is given a section of the graph and searches through it. The two processes at the bottom offer a scalable way of swapping data between the workers. The queue_store can implement an Alt to read in multiple senders to store the neighbours from each of the workers. By sending it to a second process it can return immediately to the Alt and handle the next incoming send. The Queue Sender waits on an Alt for a worker asking for data, and can then respond by removing the first set of data from its internal queue, and then check if the Store has anything to put on the Queue.

2.5.2 Implementation

The next stage was to implement the algorithm. The search works on a Compressed Row Stored (CSR) matrix and it was necessary to find how best to search this structure. A presentation on Large-Scale Graph Traversal Algorithms on Multi-core Clusters [30] shows diagrams of how
to search the CSR graph using the row and column indices in the CSR data structure to traverse the graph. A serial example of this graph and search method was constructed using the serial implementation of Breadth First search. On completing this, the serial algorithm was adapted to implement the 1D Level algorithm in serial - omitting the decomposing and sending and receiving of the neighbouring vertices. Having completed this and verified using the small test graph, the implementation was then inserted into the worker process. The final step was implementing the sending and receiving of distributed neighbours from the decomposed data.

```python
@csp.multiprocess
def graph_decomposer(to_worker, workers=1):
    graph = gr.read_graph("src/c_graph")
    #graph = gr.test_graph()
    split = graph.shape[1]/workers
    graph_chunk = [graph[0+split*nw: split*(nw+1)] for nw in xrange(workers)]
    for worker in xrange(workers):
        to_worker(graph_chunk[worker])
```

Figure 2.17: CSP diagram for PyCSP implementation of Breadth First Search graph solver

Figure 2.18: Graph Decomposer process
The graph decomposer uses methods from graph reader (gr) to read in the edge list from disk and turn it into a symmetric space matrix. This is then decomposed amongst the workers. The decomposition is in one dimension over the rows of the matrix and stored in a list. For the number of works specified as a argument to the decomposer each is sent their portion of the matrix.

```python
@csp_multiprocess
def bfs_worker(queue_put, queue_get, queue_ask, graph_in):
    graph = graph_in()
    sp_rows = graph.indptr.tolist()
    sp_col = graph.indices.tolist()
    Level = {0:[0]}
    i = 0 # level counter
    while True:
        Level[i+1] = []
        F = Level[i]
        if not F:
            break
        for u in F:
            try:
                start, end = sp_rows[u:u+2]
            except:
                break
                neighbors = sp_col[start:end]
            queue_put(neighbors)
            queue_ask('1')
            new_neighbors = queue_get()
            u_neighbors = list(set(neighbors) | set(new_neighbors))
            for v in u_neighbors:
                if v not in [u for x in Level.values() for u in x]:
                    Level[i+1].append(v)
            i +=1
    csp.retrive(queue_put, queue_get, queue_ask, graph_in)
```

Figure 2.19: Breadth first search worker
The bfs_worker is the main component of the solver, and read in the decomposed matrix. Using the row and column indices the worker will resolve the current frontier of vertices and append them to the current level. The level is incremented when the neighbours of the current value has been gathered from the other workers and recorded at the current level. This code mirrors the pseudo-code algorithm as closely as possible to try to implement the algorithm as correctly as understood.

```python
@csp.multiprocess
def queue_push(worker_put, queue_out):
    while True:
        chan, msg = csp.AltSelect(csp.InputGuard(worker_put))
        if chan:
            queue_out(msg)

@csp.multiprocess
def queue(queue_in, worker_ask, worker_get):
    queue_list = []
    while True:
        queue_list.append(queue_in())
        chan = csp.AltSelect(csp.InputGuard(worker_ask))[0]
        if chan == worker_ask:
            try:
                worker_get(queue_list.pop())
            except:
                pass

if __name__ == "__main__":
    N = 2  # number of BFS workers
    PAR = csp.Par()
    CHAN_A2Qask = csp.Channel('A')
    CHAN_A2Qget = csp.Channel('B')
    CHAN_A2Qput = csp.Channel('C')
    CHAN_Q2Q = csp.Channel('D')
    to_worker = csp.Channel('G')
    PAR.Parallel(graph_decomposer(-to_worker, workers=N),
                 queue(+CHAN_A2Qput, -CHAN_Q2Q),
                 bfs_worker(-CHAN_A2Qput, +CHAN_A2Qget, -CHAN_A2Qask, +
                 to_worker) * N)
    csp.shutdown()
```

Figure 2.20: Queue processes

Figure 2.21: Main program entry point
The Queue processes are required due to the blocking nature of CSP channels. This allows each of the workers to submit their current set of neighbours. Each is then collected and sent to the other workers. The use of the Alt allows the Queue processes to wait on a worker progressing to the point where they can send, without blocking on a slow worker. The requirement for two is to allow more than one worker to place items in the queue and then pop them from the list stored in the other process. Condensing this into one process would result in a bottleneck or process waiting for items on the queue.

Each of the processes uses a standard PyCSP parallel (as opposed to the MPI parallel) and defines each process and its channels. Multiple workers are defined using the overloaded multiply symbol to create multiple workers and their channels. This saves writing out multiple workers and connecting individual channels.

When the program is complete - the while True in the worker has ended - the worker will ask that the channels retire. This tidies up any communication still present in the other processes and the parallel can terminate. This then allows csp.shutdown - equivalent to MPI_Finalise() - to terminate any outstanding PyCSP resources and exit the program.
Chapter 3

Results

3.1 Graph500 MPI Benchmark (C version)

3.1.1 Test conditions

The graph500 specification code offers various implementations of the graph500 search. The MPI version is provided with a few example implementations of the breadth first search using different techniques in MPI such as single sided communication. The MPI implementation also offers hybrid examples using OpenMP. For this project the make file has been altered to provide the compiled instance of the sample code for ARCHER. This can be seen in Figure 3.1 and the fully supplied version in Appendix 6.1.

```bash
LDLFLAGS = 
MPICC = cc 
CFLAGS = -O3 -hfp3 -Drestrict=_restrict_ -DGRAPH_GENERATOR_MPI 
all: graph500_mpi_simple 

GENERATOR_SOURCES = ./generator/graph_generator.c 
\ ./generator/make_graph.c \ ./generator/splittable_mrg.c 
\ ./generator/utils.c 
SOURCES = main.c onedcsr.c onedcsc.c utils.c validate.c onesided.c 
onesided_emul.c 
HEADERS = common.h onedcsr.h onedcsc.h redistribute.h mpi_workarounds.h 
onesided.h 

graph500_mpi_simple: bfs_simple.c $(SOURCES) $(HEADERS) 
\ $(GENERATOR_SOURCES) $(MPICC) $(CFLAGS) $(LDLFLAGS) -o 

graph500_mpi_simple bfs_simple.c $(SOURCES) $(GENERATOR_SOURCES) -lm 

clean: 
\ -rm -f graph500_mpi_* *.o *.a
```

Figure 3.1: Modified Makefile for Graph500 test

When the code was run initially it was not able to complete the graph in a suitable time. The
scale factor was set to 26 resulting in 67,108,864 vertices and 1,073,741,824 edges. The test was run until it completed successfully resulting in a required, 128 nodes and 3072 MPI process to complete in time. One of the problems was the time taken to verify the results of each of the 64 searches. Due to time and considerations made in the development of the PyCSP version it was revised to give results for the the time taken to complete the graph at different scales for one node. Each node has 24 cores and each core will be given an MPI process.

Figure 3.2 shows the script used to launch one of the jobs. Four scales - 12, 19, 22, 26 - of the graph size were run; each job was run for a set of MPI processes and writes the results to disk.

```bash
#!/bin/bash

#PBS -N Graph-500
#PBS -l select=1
#PBS -lwalltime=00:05:0
#PBS -A d68

declare -a proc=(1 2 4 8 16 24)

# This shifts to the directory that you submitted the job from
cd $PBS_O_WORKDIR
export SKIP_VALIDATION=1
export TMPFILE=12_graph
export REUSEFILE=12_graph

for p in "${proc[@]}"
do
    aprun -n $p .graph500_mpi_simple 12 > output12-$p.out 2>&1
done
```

Figure 3.2: Graph500 sample code PBS script for one node at scale 12

The environment variables were constant for each run, turning off verification and providing a name for the file. The results of each run are tabled in Table 3.1.2 with the added times for the run of scale 26 as 128 nodes.

### 3.1.2 Graph500 Results (C version)

<table>
<thead>
<tr>
<th>SCALE</th>
<th>graph_generation</th>
<th>num_mpi</th>
<th>construct_time</th>
<th>max_time</th>
<th>stddev_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0.00730801</td>
<td>24</td>
<td>0.0139911</td>
<td>0.003735707</td>
<td>1.54E-05</td>
</tr>
<tr>
<td>19</td>
<td>0.629941</td>
<td>24</td>
<td>0.716639</td>
<td>0.0453639</td>
<td>5.62E-04</td>
</tr>
<tr>
<td>22</td>
<td>0.836143</td>
<td>24</td>
<td>1.62771</td>
<td>0.32676</td>
<td>0.00111185</td>
</tr>
<tr>
<td>26</td>
<td>12.6612</td>
<td>24</td>
<td>28.0326</td>
<td>5.39301</td>
<td>0.0111059</td>
</tr>
<tr>
<td>26</td>
<td>18.6299</td>
<td>3072</td>
<td>14.0901</td>
<td>2.88266</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 3.1: Graph500 results for one full node - 24 process- at each graph scale

* This run was the attempt at the required minimum number of MPI process needed to complete the full graph500 instance at scale 26. Verification was turned on for this run with consumed resources and was not used in the later tests.
<table>
<thead>
<tr>
<th>SCALE</th>
<th>graph_generation</th>
<th>num_mpi</th>
<th>construct_time</th>
<th>min_time</th>
<th>max_time</th>
<th>stddev_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0.0265441</td>
<td>1</td>
<td>0.012105</td>
<td>0.00293708</td>
<td>0.00299907</td>
<td>1.92E-05</td>
</tr>
<tr>
<td>12</td>
<td>0.00656796</td>
<td>2</td>
<td>0.0123849</td>
<td>0.00302696</td>
<td>0.00314403</td>
<td>1.38E-05</td>
</tr>
<tr>
<td>12</td>
<td>0.00718689</td>
<td>4</td>
<td>0.0127518</td>
<td>0.00324702</td>
<td>0.003335</td>
<td>1.66E-05</td>
</tr>
<tr>
<td>12</td>
<td>0.00722599</td>
<td>8</td>
<td>0.0136659</td>
<td>0.00351095</td>
<td>3.55E-03</td>
<td>7.18E-06</td>
</tr>
<tr>
<td>12</td>
<td>0.00718403</td>
<td>16</td>
<td>0.013803</td>
<td>0.00361109</td>
<td>0.00427318</td>
<td>8.07E-05</td>
</tr>
<tr>
<td>12</td>
<td>0.00730801</td>
<td>24</td>
<td>0.0139911</td>
<td>0.00362897</td>
<td>0.00373507</td>
<td>1.54E-05</td>
</tr>
</tbody>
</table>

Table 3.2: Graph500 results for Scale 12

<table>
<thead>
<tr>
<th>SCALE</th>
<th>graph_generation</th>
<th>num_mpi</th>
<th>construct_time</th>
<th>min_time</th>
<th>max_time</th>
<th>stddev_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>0.790883</td>
<td>1</td>
<td>2.06451</td>
<td>0.399528</td>
<td>0.410434</td>
<td>0.00156961</td>
</tr>
<tr>
<td>19</td>
<td>0.556753</td>
<td>2</td>
<td>1.39775</td>
<td>0.344795</td>
<td>0.346945</td>
<td>0.000419654</td>
</tr>
<tr>
<td>19</td>
<td>0.592343</td>
<td>4</td>
<td>1.01341</td>
<td>0.198309</td>
<td>0.20039</td>
<td>0.000521682</td>
</tr>
<tr>
<td>19</td>
<td>0.639701</td>
<td>8</td>
<td>0.844094</td>
<td>0.109326</td>
<td>0.111283</td>
<td>0.000402307</td>
</tr>
<tr>
<td>19</td>
<td>0.628806</td>
<td>16</td>
<td>0.743075</td>
<td>0.0584629</td>
<td>0.0608492</td>
<td>6.13E-04</td>
</tr>
<tr>
<td>19</td>
<td>0.629941</td>
<td>24</td>
<td>0.716639</td>
<td>0.043354</td>
<td>0.0453639</td>
<td>5.62E-04</td>
</tr>
</tbody>
</table>

Table 3.3: Graph500 results for Scale 19

<table>
<thead>
<tr>
<th>SCALE</th>
<th>graph_generation</th>
<th>num_mpi</th>
<th>construct_time</th>
<th>min_time</th>
<th>max_time</th>
<th>stddev_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>4.80895</td>
<td>1</td>
<td>22.3538</td>
<td>3.59975</td>
<td>3.66905</td>
<td>0.0115404</td>
</tr>
<tr>
<td>22</td>
<td>3.37572</td>
<td>2</td>
<td>11.2749</td>
<td>2.91507</td>
<td>2.95624</td>
<td>0.00539458</td>
</tr>
<tr>
<td>22</td>
<td>2.27344</td>
<td>4</td>
<td>6.08213</td>
<td>1.62888</td>
<td>1.6459</td>
<td>0.00310387</td>
</tr>
<tr>
<td>22</td>
<td>1.01325</td>
<td>8</td>
<td>3.31696</td>
<td>0.881078</td>
<td>0.890302</td>
<td>0.00174122</td>
</tr>
<tr>
<td>22</td>
<td>1.22642</td>
<td>16</td>
<td>2.06445</td>
<td>0.464662</td>
<td>0.470392</td>
<td>0.00138788</td>
</tr>
<tr>
<td>22</td>
<td>0.836143</td>
<td>24</td>
<td>1.62771</td>
<td>0.321799</td>
<td>0.32676</td>
<td>0.00111185</td>
</tr>
</tbody>
</table>

Table 3.4: Graph500 results for Scale 22

<table>
<thead>
<tr>
<th>SCALE</th>
<th>graph_generation</th>
<th>num_mpi</th>
<th>construct_time</th>
<th>min_time</th>
<th>max_time</th>
<th>stddev_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>98.3229</td>
<td>1</td>
<td>920.407</td>
<td>67.1938</td>
<td>67.2706</td>
<td>0.0178972</td>
</tr>
<tr>
<td>26</td>
<td>56.3777</td>
<td>2</td>
<td>390.639</td>
<td>51.6419</td>
<td>51.75</td>
<td>0.028925</td>
</tr>
<tr>
<td>26</td>
<td>44.1044</td>
<td>4</td>
<td>173.108</td>
<td>29.0679</td>
<td>29.1819</td>
<td>0.0281701</td>
</tr>
<tr>
<td>26</td>
<td>29.2613</td>
<td>8</td>
<td>102.174</td>
<td>15.8148</td>
<td>15.9112</td>
<td>0.0225621</td>
</tr>
<tr>
<td>26</td>
<td>14.6115</td>
<td>16</td>
<td>41.5264</td>
<td>8.01662</td>
<td>8.06182</td>
<td>0.00920867</td>
</tr>
<tr>
<td>26</td>
<td>12.6612</td>
<td>24</td>
<td>28.0326</td>
<td>5.329</td>
<td>5.39301</td>
<td>0.0111059</td>
</tr>
</tbody>
</table>

Table 3.5: Graph500 results for Scale 26
3.2 Python/CSP Results Comparison

3.2.1 Errors

When using the MPI version, the program cannot make use of more than one worker process. This is due to being not able to create multiple MPI channels with multiple readers and writers. A solution to this problem is shown in further work section 2.3. The program was amended to use the standard channel object and parallel. This does restrict the PyCSP program to run only on one ARCHER node. The test conditions and results to compare with graph500 were amended to account for this.

The standard PyCSP solution works, however there is an error in the reported level of a vertex. When the graph is decomposed, the adjunct matrix is given to each worker. Each worker looks up neighbours of a vertex using the row and column index of the matrix. For some nodes this changes the order in which vertices are visited and this changes the level at which they are recorded. Attempts were made to alter this, first, by decomposing the matrix by columns and, second, by slight changes in the way neighbouring vertices are sent to other workers. On testing both solutions they both failed to place some of the vertices at the correct level. This results - using the test graph of 7 vertices - with 2 of the 7 recorded in a level greater than they should have been. While this is not ideal and limits the program to only searching from 1 key - to ensure the workers do own the right starting vertex - it does however show that all vertexes were visited and that this part of the breadth first search is working.

3.2.2 Test Conditions

The test was run using the python2.7 interpreter compiled by Cray on ARCHER. It uses numpy (version 1.8.0), scipy (version 0.13.3) and mpi4py (version 1.3.1). The test is run by allocation one python instance for one node, with a run for each increment of workers - 1, 2, 4, 8, 16, 24 - and timed. The times record the overall time to complete, and the time to load and build the adjunct matrix. The graph generation is not recorded as it uses the same graph as used by the graph500 results.

3.2.3 Execution Time

Table 3.6 show the execution time to read and build the adjunct matrix into a python data structure, and time taken by the program to search the graph. Each scale of graph was tested in a similar manner to the Graph500 test seen in Section 3.1.2

3.2.4 Errors during results

It was found on the graph scale of 12 that when trying to run using 24 workers that the graph search did not terminate. Taking the program back to the development and running a comparison on ARCHER, it was found through print statements that some workers had no neighbours to search through. This would indicate that the decomposition of the graph in relation to the number of workers results in some workers without adequate work to do. This causes a process to
block preventing termination, or finish with the possibility of some outstanding communication that prevents the program from terminating correctly.

There is an absence of results for the grater scales. During testing the graph was run with a scale level of 19. The length of time take to work was longer than 2 hours to complete. This prompted the changes made to the graph reader module. This is discussed in Section 3.2.6. Tested on the development workstation the amended version could complete 1 64th of the dataset in 30 seconds. A full run of would take a possible 30 minutes to build the sparse matrix. It is predicted that the use of multi channels would have allowed the decomposition of the graph and the sparse matrix constructed on each of the workers. 64 workers would have been required and due to the limit of 24, the grater sized graphs were not tested.

### 3.2.5 Length of Program

Python’s syntax allows the user to express the work to be done in few key words and lines than the C version. In some cases this is due to the programmer not needing to preallocate memory or define types for variables. Python makes assumptions on variables that if this is the first time the variable is used, and it is being assigned a list, then this variable is allocated and typed as a list until it is explicitly cast as another variable or overwritten. With consideration as to the correctness of the implementation, the number of lines for each is shown in Table 3.7.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Total lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>C bfs.c</td>
<td>175</td>
</tr>
<tr>
<td>Pycsp bfs_worker</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 3.7: Line counts for each implementation

The C version is longer due to the management of the data structures and the queues for incoming messages from neighbours. There is also the use of macros to try and reduce the complexity in iterating over the vertices. These macros help determine if a vertex has been visited on the bitmap of nodes. The python version tests for visitation using a list comprehension over the level dictionary. Vertices are set by adding into this structure and the test is the compression loop over the dictionary. This is a more expensive operation than checking if a bit has been set, but results in a shorter and more visibly clear program.
### Table 3.8: Execution time with initial graphreader.py implementation

<table>
<thead>
<tr>
<th>SCALE</th>
<th>construction_time</th>
<th>stddev_time</th>
<th>num_workers</th>
<th>completion_time</th>
<th>stddev_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>3.704339345</td>
<td>1.165233873</td>
<td>1</td>
<td>22.62175504</td>
<td>1.615372901</td>
</tr>
<tr>
<td>12</td>
<td>3.361498356</td>
<td>1.576420557</td>
<td>2</td>
<td>3.463044008</td>
<td>1.55058553</td>
</tr>
<tr>
<td>12</td>
<td>2.445601702</td>
<td>0.00489394</td>
<td>4</td>
<td>2.542200724</td>
<td>0.00489394</td>
</tr>
<tr>
<td>12</td>
<td>2.43635048</td>
<td>0.004975978</td>
<td>8</td>
<td>2.582115332</td>
<td>0.003894847</td>
</tr>
<tr>
<td>12</td>
<td>2.4907266</td>
<td>0.066988261</td>
<td>16</td>
<td>2.707217693</td>
<td>0.071827643</td>
</tr>
<tr>
<td>12</td>
<td>2.446461678</td>
<td>0.004840208</td>
<td>24</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**3.2.6 Performance Pitfalls**

Parts of the python program performed badly, through external forces and choices made in parts of the program. One particularly slow component of the program is graph generation. This file can be seen in the Appendix 6.3 & 6.4. For this program is was loaded by one of the PyCSP processes and then a Compressed Sparse Row (CSR) matrix slice is sent to the workers. The edge list is read in from disk and converted into a Numpy array and then converted into a python list. This is due to the initial implementation to solve duplicates and build the adjunct matrix. This is implemented naively and has two loops, the first builds two lists of each start and end vertex while deleting elements where the vertex appears twice. The second loop works on a python nested list of zeros, building the sparse matrix. This performs badly as it requires two liner loops over a large data structure and allocates more memory than is required.

Another performance problem was sorting the list of edges. To index the matrix correctly the edges list was sorted so that the start vertex was in order followed by the end vertex. This does simplify and prevent the creation of the adjunct matrix from jumping in memory to set each value. However, as this was a python list, it used the standard python sort. For a large list - and this was made clear in trying to load the scale 19 graph - this caused a dramatic slow down. A scale 19 graph is 128MB in size. Python implements a hybrid insertion and merge sort, and should scale at worst as N log N. However this performed worse than the Numpy merge sort when reimplemented. This is possibly due to, the original version’s data structure performing poorly in memory as merges occur during the sort. The Numpy version performs an in place merge sort on the data structure. This structure is an optimised Numpy array, allowing the sort to take advantage of optimisations in the data structure and in the sorting algorithm itself.

The last improvement to this part of the program is to modify the creation of the adjunct matrix. It is a poor decision to unpack the edge list to create two separate lists. This can be better implemented by python creating two references to each inner element over a Numpy array. For the number of edges each edge vertex can be referenced when used to build the sparse matrix.
Chapter 4

Conclusions

The goal of creating a Graph500 implementation in using Python and CSP has been varied in success. The final program did perform within the expected parameters for Python, and was noted at the beginning to have the expected performance lower than C. A full implementation of Graph500 is expected to take longer to produce than the project’s time allocation, and would not perform as well as the sample code. However, programming in Python for High Performance does have advantages. Python’s concise and quick access to library lowers the barrier to a serviceable program. Python’s ability to process large amounts of data were restricted by the desire and implementation of the PyCSP solver and missing infrastructure components. A program that scaled beyond one node is expected to have completed the scale 19 graph in a reasonable amount of time. However the program did perform within expectation for the smaller graph sizes.

During development the code was moved between a development machine to ARCHER without changes in compiler or development tools. This makes Python for high performance programming less cumbersome and portable because any system that has Python installed can run the code. There are exceptions for libraries, but the use of tools such as virtualenv allow for exotic library to be installed without having to be supplied by the system provider.

PyCSP greatly simplifies the interaction with the threading and communication primitives in Python. The overall required syntax of decorators, channel objects, and process construction through the Parallel, to create a PyCSP program is small and concise. This results in only a few concepts for the programmer to learn and moves them on quicker to consider how best to implement their system. Python’s quick prototyping ability should help aid the programmer in designing their system quickly and redesigning the system. With a full implementation of PyCSP for ARCHER, this system should have been able to execute at the required scale to complete the search of the graph. Further work on PyCSP for supercomputers would allow PyCSP to be put to further tests in scientific applications.

CSP itself for describing parallel systems helps describe the extent of the system from the beginning. It is still helpful to have a serial program, but the parallel system can be defined by breaking up tasks into processes and moving the data over channels. The resulting designs are independent of implementation and the use of diagrams helps visualise to the programmer what events should occur. For this project, it helped find out what was required when implementing the infrastructure. For example, a process taking in multiple channels will require the use of an Alternative. This lead to the implementation of the Alt to support the distributed channels. The diagrams created for this project can be used to program a solution to graph500 in a different
CSP implementation without impact on the overall system. However, the graph does not protect against overlooking features. While multiple channels was implemented in the current version of PyCSP, it was not apparent until implementing the program that they would be required for the distributed version. A distributed version could have been developed through the use of point to point channels for new workers. However, this would have been cumbersome to program and increased the complexity of the program. Alternative solutions were considered but, when described using the CSP diagram, presented flaws with distributing data, or communication bottlenecks when sharing neighbour vertexes.

The CSP model had limited the problems with parallelism and communication during this project, and the resulting problems were restricted to errors in programming, implementation of features and the consequent capabilities of the system due to these factors. For future experiments CSP seems to be useful for High Performance Commuting, and a suitable implementation of the model would add value to programming for large supercomputers. Strict design rules and a suitable implementation could reduce some of the complexity in programming, and allow other languages to fully implement a suitable interface to provide parallel programs.
Chapter 5

Further Work

5.1 Multi Channels

For PyCSP to scale beyond one node on ARCHER would have required a correct implementation of multiple-ended channels. The implementation in this project shows the scope and indicates that further work is needed. The implemented version for this project was not made in time to be used and still has unresolved implementation errors. The channels relationship with each other is information that can be resolved through parsing the list of processes in the Parallel. A newer implementation could understand this and issue the correct collective, or know where sends should be sent and in what manner. The intent of the programmer would need to be expressed to help choose the collective, where some may wish data to be broadcasted and others may require a scatter of data used in decomposition.

5.2 Distributed Processes

For PyCSP to work on ARCHER, processes are distributed using MPI. This is portable for machines that have a implementation of MPI and the MPI binding library for Python. This places a dependency on the PyCSP library. There is scope for the processes to be disseminated independent of MPI. This would require considerations as well to how distributed channels are implemented, however work to assess the requirements should be considered. Processes would need to know the node they are to run on, and a data structure similar to MPI_COMM_WORLD, to allow processes to know where they live, and allow channels to connect to each other without the programmers prior knowledge.

5.3 PyCSP upgrade

PyCSP currently sports python 2.7 but is missing support for python 3. It is important to keep libraries up to date with the changes in the language and python 3 will continue to improve and add support for features that may be relevant to High Performance Computing. An attempt to see to what extent an upgrade would change indicated that a change in the interface to threading and multiprocess would be required.
5.4 CSP implementations

Using CSP for high performance programming is well within the reach of some implementations. Opportunities to explore the effectiveness and constraints in performance with traditional parallelisation schemes should be considered. CSP could be used effectively and help implement systems that can flexibly change the implementation of processes without impacting on the integrity of the program.

5.5 CSP design in other parallel implementations

Using CSP to design programs using traditional tools should be considered. Abstracting away the technical details and using CSP to design the intended system, could help spot problems in communication and work distribution before implementation. While some of the concepts in CSP may not be applicable, drawing diagrams of systems and working through potential designs may help programmers avoid mistakes. These can arise from moving data in badly performing ways or creating tasks that are too small or too large to exploit the parallel infrastructure.
6.1 Graph500 Makefile

LDFLAGS = #--fopenmp -g # -g -pg
MPICC = cc #mpicc
CFLAGS = -O3 -hfp3 -D restrict=___restrict___ -DGRAPH_GENERATOR_MPI

all: graph500_mpi_simple graph500_mpi_one sided graph500_mpi_replicated
graph500_mpi_replicated_csc
graph500_mpi_custom

GENERATOR_SOURCES = .. / generator / graph_generator.c .. / generator / make_graph.
c .. / generator / splittable_mrg.c .. / generator / utils.c
SOURCES = main.c oned csr.c oned_csc.c utils.c validate.c oned_csc c
oned_csc_emul.c
HEADERS = common.h oned csr.h oned_csc.h redistribute.h mpi_workarounds.h
oned_csc.h

graph500_mpi_simple: bfs_simple.c $(SOURCES) $(HEADERS)
$(GENERATOR_SOURCES) $(MPICC) $(CFLAGS) $(LDFLAGS) -o
graph500_mpi_simple bfs_simple.c $(SOURCES) $(GENERATOR_SOURCES) -lm

graph500_mpi_one sided: bfs_one sided.c $(SOURCES) $(HEADERS)
$(GENERATOR_SOURCES) $(MPICC) $(CFLAGS) $(LDFLAGS) -o
graph500_mpi_one sided bfs_one sided.c $(SOURCES) $(GENERATOR_SOURCES) -lm

graph500_mpi_replicated: bfs_replicated.c $(SOURCES) $(HEADERS) $(
GENERATOR_SOURCES)
$(MPICC) $(CFLAGS) $(LDFLAGS) -o graph500_mpi_replicated
bfs_replicated.c $(SOURCES) $(GENERATOR_SOURCES) -lm

graph500_mpi_replicated_csc: bfs_replicated_csc.c $(SOURCES) $(HEADERS) $(
GENERATOR_SOURCES)
$(MPICC) $(CFLAGS) $(LDFLAGS) -o graph500_mpi_replicated_csc
bfs_replicated_csc.c $(SOURCES) $(GENERATOR_SOURCES) -lm

clean:
-rm -f graph500_mpi_* .o *.a
This version is the traditional level-synchronized BFS using two queues. A bitmap is used to indicate which vertices have been visited. Messages are sent and processed asynchronously throughout the code to hopefully overlap communication with computation.

```c
void run_bfs(int64_t root, int64_t* pred) {
    const size_t nlocalverts = g.nlocalverts;

    /* Set up the queues. */
    int64_t* restrict oldq = g.oldq;
    int64_t* restrict newq = g.newq;
    size_t oldq_count = 0;
    size_t newq_count = 0;

    /* Set up the visited bitmap. */
    const int ulong_bits = sizeof(unsigned long) * CHAR_BIT;
    int64_t visited_size = (nlocalverts + ulong_bits - 1) / ulong_bits;
    unsigned long* restrict visited = g.visited;
    memset(visited, 0, visited_size * sizeof(unsigned long));
    #define SET_VISITED(v) do { visited[VERTEX_LOCAL((v)) / ulong_bits] |= (1UL << (VERTEX_LOCAL((v)) % ulong_bits)) ; } while (0)
    #define TEST_VISITED(v) ( ( visited[VERTEX_LOCAL((v)) / ulong_bits] & (1UL << (VERTEX_LOCAL((v)) % ulong_bits))) != 0 )

    /* Set up buffers for message coalescing, MPI requests, etc. for communication. */
    const int coalescing_size = 256;
    int64_t* restrict outgoing = g.outgoing;
    size_t* restrict outgoing_counts = g.outgoing_counts;
    MPI_Request* restrict outgoing_reqs = g.outgoing_reqs;
    int* restrict outgoing_reqs_active = g.outgoing_reqs_active;
    memset(outgoing_reqs_active, 0, size * sizeof(int));
    int64_t* restrict recvbuf = g.recvbuf;
    MPI_Request recvreq;
    int recvreq_active = 0;

    /* Termination counter for each level: this variable counts the number of ranks that have said that they are done sending to me in the current level. This rank can stop listening for new messages when it reaches size. */
    int num_ranks_done;

    /* Set all vertices to "not visited." */
    { size_t i; for (i = 0; i < nlocalverts; ++i) pred[i] = -1;}

    /* Mark the root and put it into the queue. */
    if (VERTEX_OWNER(root) == rank) {
        SET_VISITED(root);
        pred[VERTEX_LOCAL(root)] = root;
        oldq[oldq_count++] = root;
    }
    #define CHECK_MPI_REQS
    /* Check all MPI requests and handle any that have completed. */
    { do {
        /* Test for incoming vertices to put onto the queue. */
```
while (recvreq_active) {
    int flag; 
    MPI_Status st; 
    MPI_Test(&recvreq, &flag, &st); 
    if (flag) { 
        recvreq_active = 0; 
        int count; 
        MPI_Get_count(&st, MPI_INT64_T, &count); 
        /* count == 0 is a signal from a rank that it is done sending to me 
        * (using MPIs non-overtaking rules to keep that signal after all 
        * "real" messages. */ 
        if (count == 0) { ++num_ranks_done; 
            int j; 
            for (j = 0; j < count; j += 2) { 
                int64_t tgt = recvbuf[j]; 
                int64_t src = recvbuf[j + 1]; 
                /* Process one incoming edge. */ 
                assert (VERTEX_OWNER(tgt) == rank); 
                if (!TEST_VISITED(tgt)) { 
                    SET_VISITED(tgt); 
                    pred[VERTEX_LOCAL(tgt)] = src; 
                    newq[newq_count++] = tgt; 
                } 
            } 
            /* Restart the receive if more messages will be coming. */ 
            if (num_ranks_done < size) 
                MPI_Irecv(recvbuf, coalescing_size * 2, MPI_INT64_T, 
                    MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &recvreq); 
                recvreq_active = 1; 
            } 
        } else break; 
    } 
    /* Mark any sends that completed as inactive so their buffers can be 
    * reused. */ 
    int c; 
    for (c = 0; c < size; ++c) { 
        if (outgoing_reqs_active[c]) { 
            int flag; 
            MPI_Test(&outgoing_reqs[c], &flag, MPI_STATUS_IGNORE); 
            if (flag) outgoing_reqs_active[c] = 0; 
        } 
    } 
} while (0) 

while (1) { 
    memset(outgoing_counts, 0, size * sizeof(size_t)); 
    num_ranks_done = 1; /* I never send to myself, so I'm always done */ 
    /* Start the initial receive. */ 
    if (num_ranks_done < size) 
        MPI_Irecv(recvbuf, coalescing_size * 2, MPI_INT64_T, MPI_ANY_SOURCE, 
            0, MPI_COMM_WORLD, &recvreq); 
        recvreq_active = 1; 
}
/* Step through the current level's queue. */
size_t i;
for (i = 0; i < oldq_count; ++i) {
    CHECK_MPI_REQS;
    assert (VERTEX_OWNER(oldq[i]) == rank);
    assert (pred[VERTEX_LOCAL(oldq[i])] >= 0 && pred[VERTEX_LOCAL(oldq[i])]) < g.nglobalverts);
    int64_t src = oldq[i];
    /* Iterate through its incident edges. */
    size_t j, j_end = g.rowstarts[VERTEX_LOCAL(oldq[i]) + 1];
    for (j = g.rowstarts[VERTEX_LOCAL(oldq[i])]; j < j_end; ++j) {
        int64_t tgt = g.column[j];
        int owner = VERTEX_OWNER(tgt);
        /* If the other endpoint is mine,
         * update the visited map, predecessor map,
         * and next-level queue locally; otherwise, send the target and
         * the current vertex (its possible predecessor)
         * to the target's owner.
         */
        if (owner == rank) {
            if (!TEST_VISITED(tgt)) {
                SET_VISITED(tgt);
                pred[VERTEX_LOCAL(tgt)] = src;
                newq[newq_count++] = tgt;
            }
        } else {
            while (outgoing_reqs_active[owner]) CHECK_MPI_REQS; /* Wait for buffer to be available */
            size_t c = outgoing_counts[owner];
            outgoing[owner * coalescing_size * 2 + c] = tgt;
            outgoing[owner * coalescing_size * 2 + c + 1] = src;
            outgoing_counts[owner] += 2;
            if (outgoing_counts[owner] == coalescing_size * 2) {
                MPI_Isend(&outgoing[owner * coalescing_size * 2],
                           MPI_INT64_T, owner, 0, MPI_COMM_WORLD, &outgoing_reqs[owner]);
                outgoing_reqs_active[owner] = 1;
                outgoing_counts[owner] = 0;
            }
        }
    }
    /* Flush any coalescing buffers that still have messages. */
    int offset;
    for (offset = 1; offset < size; ++offset) {
        int dest = MOD_SIZE(rank + offset);
        if (outgoing_counts[dest] != 0) {
            while (outgoing_reqs_active[dest]) CHECK_MPI_REQS;
            MPI_Isend(&outgoing[dest * coalescing_size * 2], outgoing_counts[dest],
                       MPI_INT64_T, dest, 0, MPI_COMM_WORLD, &outgoing_reqs[dest]);
            outgoing_reqs_active[dest] = 1;
            outgoing_counts[dest] = 0;
        }
    }
    /* Wait until all sends to this destination are done. */
    while (outgoing_reqs_active[dest]) CHECK_MPI_REQS;
    /* Tell the destination that we are done sending to them. */
    MPI_Isend(&outgoing[dest * coalescing_size * 2], 0, MPI_INT64_T, dest,
              0, MPI_COMM_WORLD, &outgoing_reqs[dest]); /* Signal no more sends */
while (outgoing_reqs_active[dest]) CHECK_MPIREQS;

} /* Wait until everyone else is done (and thus couldn't send us any more */
while (num_ranks_done < size) CHECK_MPIREQS;

/* Test globally if all queues are empty. */
int64_t global_newq_count;
MPI_Allreduce(&newq_count, &global_newq_count, 1, MPI_INT64_T, MPI_SUM, MPI_COMM_WORLD);

/* Quit if they all are empty. */
if (global_newq_count == 0) break;

/* Swap old and new queues; clear new queue for next level. */
{ int64_t* temp = oldq; oldq = newq; newq = temp;}
oldq_count = newq_count;
newq_count = 0;

return csr_matrix(adj_matrix)

6.3 First implementation of GraphReader.py

def read_graph(file_name=os.environ.get('REUSEFILE'), num_edges=12):
  nedges = int(16*math.pow(2,num_edges))
  N = int(math.pow(2,num_edges))
  graph = np.fromfile(file_name, dtype=np.int64)
  graph = graph.reshape(nedges,2)
  #print graph.shape
  py_graph = graph.tolist()
  py_graph.sort()

  edges_a = []
  edges_b = []
  for edge in py_graph:
    if edge[0] == edge[1]:
      py_graph.remove(edge)
      edges_a.append(edge[0])
      edges_b.append(edge[1])

  adj_matrix = [[0 for i in xrange(N)] for j in xrange(N)]
  for i in xrange(len(edges_a)):
    adj_matrix[edges_a[i]][edges_b[i]] = 1
    adj_matrix[edges_b[i]][edges_a[i]] = 1

  return csr_matrix(adj_matrix)
6.4 Improved implementation of GraphReader.py

```python
def read_graph(split, file_name=os.environ.get('REUSEFILE'), num_edges=12):
    nedges = int(16*math.pow(2, num_edges))
    N = int(math.pow(2, num_edges))
    graph = np.fromfile(file_name, dtype=np.int64)
    graph = graph.reshape(nedges, 2)
    graph.sort(axis=0, kind='mergesort')
    duplicates = (graph[:, 0] == graph[:, 1])
    graph = np.delete(graph, np.where(duplicates), axis=0)
    adj_matrix = [[0 for i in xrange(N)] for j in xrange(N)]
    for start, end in graph:
        adj_matrix[start][end] = 1
        adj_matrix[end][start] = 1
    return csr_matrix(adj_matrix)
```

50
Bibliography


