Performance of the Distributed CPA Protocol and Architecture on Traditional Networks

Kevin CHALMERS
School of Computing, Edinburgh Napier University,
Edinburgh, EH10 5DT, UK

Abstract. Performance of communication mechanisms is very important in distributed systems frameworks, especially when the aim is to provide a particular type of behaviour across a network. In this paper, performance measurements of the distributed Communicating Process Architectures networking protocol and stack is presented. The results presented show that for general communication, the distributed CPA architecture is close to the baseline network performance, although when dealing with parallel speedup for the Mandelbrot set, little performance is gained. A discussion into the future direction of the distributed CPA architecture and protocol in relation to occam-π and other runtimes is also presented.

Keywords. JCSP, CSP for .NET, networking, distributed systems.

Introduction

Communicating Processes for Java (JCSP) [1] is a Java library implementation of Hoare's CSP model of concurrency [2]. Recent work on JCSP has focused on refining and improving the library [3], and improving distributed systems capabilities [4]. In this paper, ongoing work allowing inter-framework communication (for example .NET to Java distributed CPA behaviour) is presented, looking at general network performance benchmarks of the new protocol and architecture.

The rest of this paper is broken up as follows. In Section 1, the background to the ongoing work with the Distributed CPA Architecture is presented. In Section 2, the approach used to gather the experimental data presented is given and reasoning behind the results provided. Section 3 briefly discusses a new implementation of CSP for .NET [5] which has been used in conjunction with JCSP to provide the results presented. Section 4 presents the results for the experiments undertaken, and finally in Section 5 conclusions and future work is presented.

1. Background

The Distributed CPA Protocol and Architecture [4] was developed to overcome problems discovered in the original JCSP Networking implementation [6]. In particular, two areas were addressed:

- The use of serialised objects for general messaging, leading to incompatibilities with other frameworks.
- Resource usage being high, particularly for the number of processes in operation.

The protocol underpinning the new implementation of JCSP networking was developed to allow inter-framework communication, and the updated framework refined and reduced the processes in use. This work led to the JCSP networking functionality being very lightweight in comparison to the original version.

To continue work enabling inter-framework communication, a number of smaller projects have been initiated looking at integrating other CSP inspired libraries and runtimes by implementing the new network architecture and protocol in PyCSP [7], C++ CSP [8] [9] and occam-π, leading to primitive communication between JCSP, PyCSP and occam-π. However, at this stage, only fully realised implementations of the networking protocol and architecture has been implemented in JCSP and CSP for .NET.

1.1 Distributed CPA Frameworks

JCSP networking is not the only implementation of distributed CPA architectures. JCSP networking itself is based on the T9000 virtual channel model for distributed CPA communications [10]. occam-π has had two separate networking capabilities developed. The first was pony [11], which emulated the same T9000 virtual channel model to a certain degree. More recent work has led to the development of a PyCSP to occam-π communication mechanism [12].

C++CSP has also featured a networking implementation [13], although this has not seen work in recent years. However, the C++CSP library itself provides interest as a library to implement a native version of the distributed CPA protocol and architecture that other runtimes can hook into. This possibility will be discussed at the end of this paper.

2. Approach

In this section, the approach that will be taken to analyse the general performance of the distributed CPA architecture shall be discussed. From a general performance point of view, we are most interested in network metrics, in particular latency and bandwidth.

2.1 Network Performance

Network latency provides a value for the general overhead of sending a single packet of data using the communication mechanism under test. It is essentially the time taken for a single packet of data to travel from the host machine to the destination. The standard method of gathering latency information is to use a ping test, and halving the time taken for a ping round-trip message.

Network bandwidth is the measure of the data throughput rate of the communication mechanism. Typically, this is measured in Mbit/second. To test bandwidth, packets of various sizes are sent across the network, and the time taken to send the data is used to determine the data throughput.

Ideally, any communication mechanism for distributed systems should have close to the general performance of the network. Therefore, baseline network performance is also gathered, and then compared to the results gathered for the distributed system communication mechanism. There may be a slight overhead for latency due to extra message headers and other communication information, but bandwidth should be similar.

2.2 Mandelbrot

To allow analysis of how well the distributed CPA architecture copes with parallel

problems, generation of the Mandelbrot set at various levels of detail has also been performed. There are two approaches that can be taken here. Firstly, breaking a single generation of the Mandelbrot set into smaller pieces will enable an analysis of the distributed CPA architecture when breaking down a single problem. Secondly, scaling the resolution of the Mandelbrot set will enable an analysis of scaling the problem to the number of processors in operation in the system.

2.3 JCSP and CSP for .NET 2.0

To provide a reasonable analysis on the performance of the distributed CPA architecture, tests were also performed using JCSP and CSP for .NET 2.0. These tests should ensure that it is the performance of the architecture and communication mechanism itself that is analysed. To avoid general overheads that are either Java or .NET dependent, object serialisation is kept to a minimum. The Mandelbrot experiments require very simple object serialisation, but nothing that should affect the comparison.

As the experiments incorporate CSP for .NET 2.0, a general discussion on the changes made since the previous edition [5] shall be discussed in the following section.

3. CSP for .NET 2.0

In this section, the new features of CSP for .NET are introduced. The new version of CSP for .NET is an update of the existing library [5] with changes to channel creation, the addition of primitive and reference passing channels, and the inclusion of networking capabilities to allow communication with other distributed CPA frameworks.

3.1 Channel Creation

Channel creation in CSP for .NET 2.0 no longer utilises factory based methods. For example, in Java using JCSP a channel is normally created using the `Channel` factory object:

```
One2OneChannel chan = Channel.one2one();
```

CSP for .NET 2.0 uses standard `new` calls to create channel objects:

```
One2OneChannel<object> chan = new One2OneChannel<object>();
```

Internally, CSP for .NET now has only one channel object type that is exposed through the necessary interfaces to provide the expected external functionality (Any2One, Any2Any, One2One, One2Any). This makes the code base easier to manage as core functionality is encapsulated in a single class.

3.2 Primitive and Reference Passing Channels

Unlike Java, .NET treats everything as an object with the exception of `IntPtr` which represents a pointer to native memory. .NET does distinguish between primitives (`struct`, or value data types) and references (`class` data types). As such, it is possible to create two separate channel types within CSP for .NET, one for dealing with value types, and one for dealing with reference types:

```
One2OneChannel<object> chan = new One2OneChannel<object>();
One2OneChannelPrimitive<int> chan
        = new One2OneChannelPrimitive<int>();
```

The two channel types provide almost identical interfaces, except when it comes to writing to them. The primitive channel type uses a standard method call:

```
int n = 5;
output.Write(n);
```

.NET will create a copy of the value `n` and use this in the write operation. This is standard copy semantics.

For reference channel types, the value passed into the `Write` call is a reference to the reference rather than a copy of the reference:

```
object obj = new object();
output.Write(ref obj);
// obj is now invalid. obj == null
```

Unlike Java, .NET has C++-like reference passing semantics. Therefore, we can modify the value stored in the reference and set it to `null` in the `Write` operation. This provides semantics similar to mobiles in occam-π, although not as rigorous. The addition of this functionality was to try and reduce as far as possible poorly managed object referencing in CSP for .NET applications.

Forcing single ownership can be achieved in other manners, such as having a `Mobile<T>` type. Currently, the CSP for .NET implementation enforces the above behaviour, and other techniques to allow global ownership of information can be gained by using standard object-orientation approaches.

3.3 Networking Capabilities

The JCSP Networking stack and protocol have been implemented in CSP for .NET, allowing communication between Java and .NET using distributed CPA semantics. The architecture is identical to that described in the previous work on JCSP Networking [4]. Abstractly, the two frameworks operate identically and can communicate with one another as if they were both the same framework. Inter-framework communication was one of the fundamental goals of abstracting the networking functionality of JCSP.

4. Results

In this section results on the performance of the CPA networking architecture shall be presented. This is the first presentation of results using the new CPA networking architecture and protocol in a standard wired Ethernet environment. Previously presented results [4] focused on mobile device communication over a wireless network. The results presented here focus on performance on machines in line with standard distributed systems.

4.1 Experimental Framework

The experimental results presented were gathered on a standard 100 Mbit/s switched Ethernet network using a number of networked nodes. The networked nodes have the

following specifications:

CPU – Intel Core Duo E8400 3.0 GHz (no hyper-threading)
Memory – 2 GB
Operating System – Microsoft Windows 7 32-bit
Software – .NET 3.5, Java 6

The machines each have two active cores in their processor, but no hyper-threading. This provides an ideal thread / process configuration of two per node. No process or thread affinity was used in the experiments, context switching and load balancing being handled by the operating system.

4.2 Network Performance

In this section, the results gathered using the experimental framework for general network performance are presented. There are two attributes that we are concerned with – latency and throughput. Both standard network performance and CSP for .NET performance data are presented. JCSP networking performance data is similar to CSP for .NET performance, and therefore not presented specifically in the following charts.

For the Distributed CPA architecture, two types of network performance have been measured – synchronous and asynchronous. Synchronous communication has an acknowledgement message sent from the input end of the channel to the output end, requiring another network message. Asynchronous communication has no acknowledgement, providing close to standard network communication behaviour. Results were also gathered using a raw data filter [4], meaning that no serialisation is involved in message sending.

For all tests conducted, TCP/IP communication is used. UDP is currently not implemented in JCSP or CSP for .NET networking due to non-guaranteed message delivery.

4.2.1 Ping and Latency

Latency is the standard measure of time for a single data packet from the sending location. The standard method of measuring network latency is to use a network ping. From a distributed system point of view, we are interested in the time taken for a standard small packet (1 byte of data) to travel from the host to the client and back again, comparing the time a normal network connection ping takes in comparison to the system under test. The latency time is then half the ping time for the system. The results presented are taken from the average figure of 10^{10} actual ping operations. Figure 1 presents the results gathered for the ping tests. The Network results are not taken from ICMP ping but a ping program written in C#.

Figure 1 shows that there is little latency difference between Network traffic, and CSP Synchronous traffic. The 0.1 ms difference in ping time will be due to the synchronisation packet being sent. Between Network traffic and CSP Asynchronous traffic, there is little difference in performance, showing that for simple data the new distributed CPA architecture and protocol provide equivalent performance to standard network communication, with only a synchronisation packet causing some slow down.

For the network used in the experimental framework, the latency is 0.35 ms. This network latency time is the time taken to acknowledge a packet, and is therefore the general overhead of a simple message for the distributed CPA architecture.

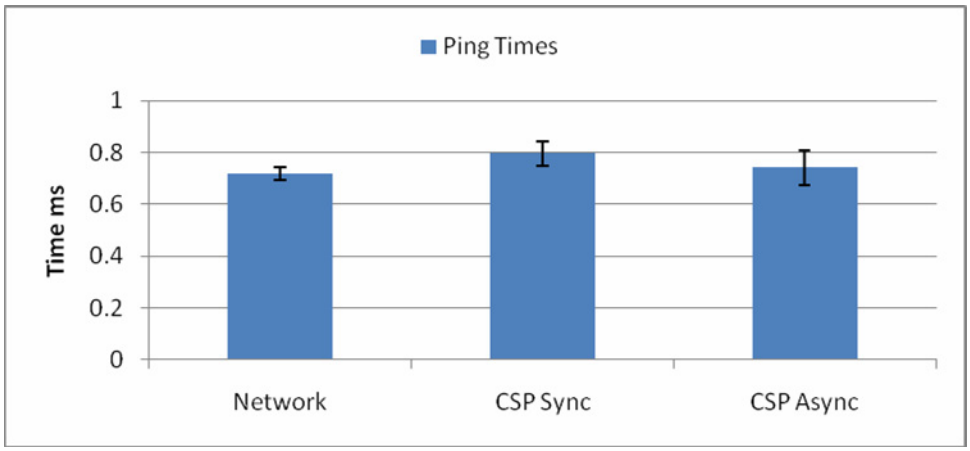

Figure 1: Ping Times

4.2.2 Throughput

Throughput provides the bytes / second performance metric of the network, and is a different indication of performance than network latency. For these results, various data packet sizes were sent over the network from the host machine to the client, with a final handshake communication performed to ensure that the data had been received. Two sets of data have been gathered. The first set of results is for sending the data from the host to the client. The second set of results shows the time taken to send the data packet and receive it back.

For these experiments, data packet sizes ranged from 1000 to 10000 bytes with an increment of 1000 bytes, and also 10000 bytes to 100000 bytes with an increment of 10000 bytes. The results presented are for the median of 10^8 actual send or send-receive operations.

Figure 2 and Figure 3 present the results for sending data packets from the host to the client. As can be seen, asynchronous communication using the CPA networked architecture meets the baseline performance of the network, except for a dip in performance at 9000 and 10000 bytes. A fuller discussion on possible reasons for occasional dips in performance has been discussed in previous work [4]. Due partially to the slight overhead of a CPA network packet (11 bytes), the amount of data sent causes extra packets to be sent, which can cause visible performance drops for small data sizes.

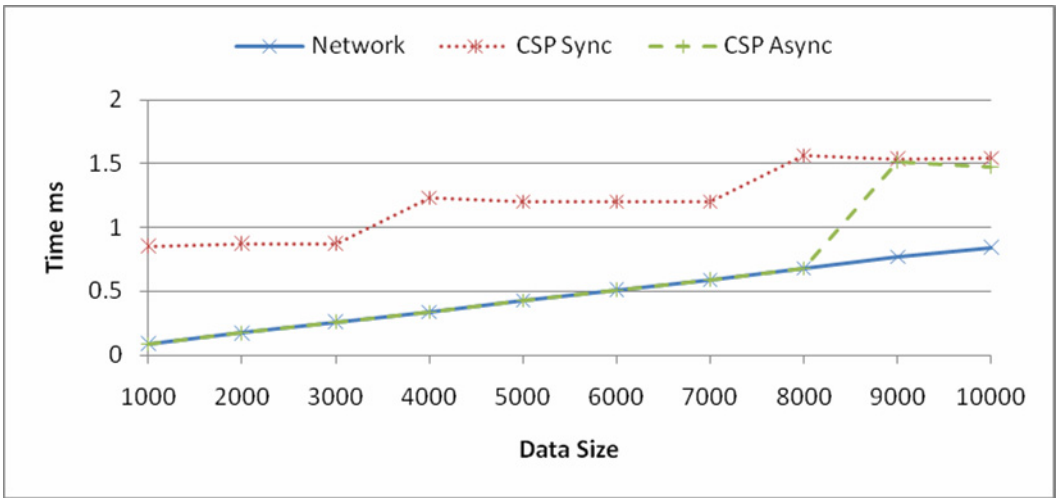

Figure 2: Sending Times 1000 to 10000

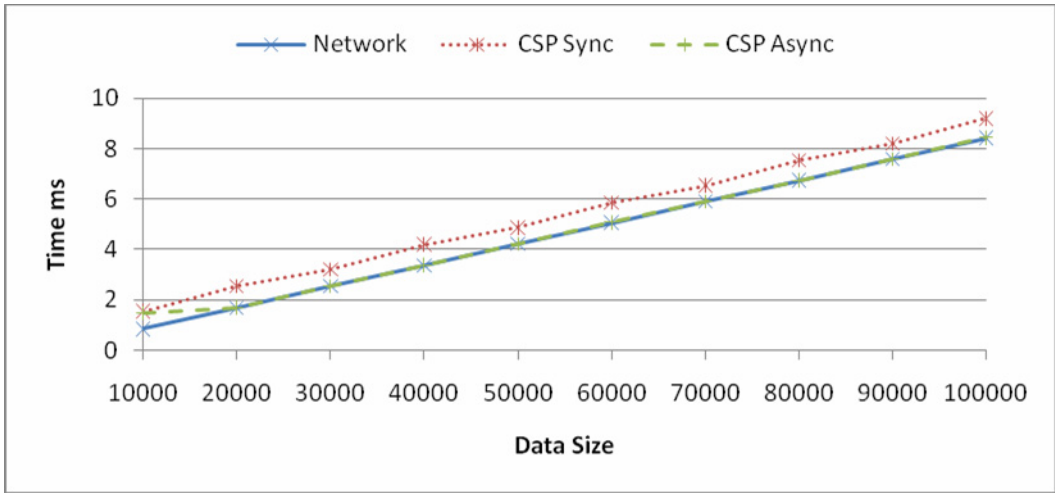

Figure 3: Sending Times 10000 to 100000

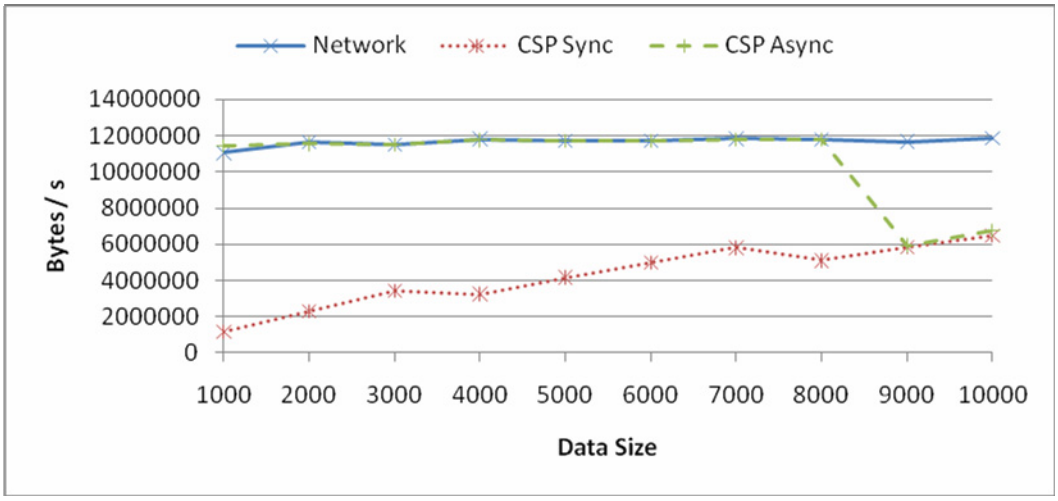

Figure 4: Sending Throughput 1000 to 10000

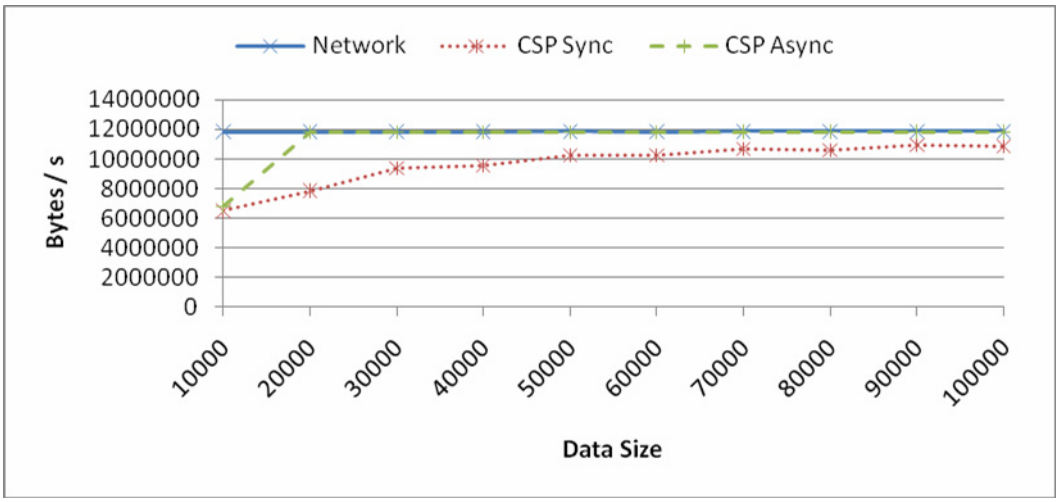

Figure 5: Sending Throughput 10000 to 100000

For synchronous communication, data takes approximately 0.5 ms longer than standard network communication. Due to the network latency of approximately 0.35 ms, this is an expected drop in performance.

Figure 4 and Figure 5 present the throughput of the network for the different experiments. This bandwidth is the value in which we are more interested. As can be seen,

for both the baseline network and asynchronous CPA communication the throughput reaches approximately 1.2×10^7 bytes per second (approximately 96 Mbit/s) almost immediately. There is the dip in performance in the asynchronous results at data sizes 9000 and 10000 as shown in Figure 2.

Synchronous results show a more gradual trend towards a tailing off at 1.1×10^7 bytes per second (approximately 88 Mbit/s). This is a dip in performance, and if the distributed application to be developed relies on raw data throughput, safe use of asynchronous communication mechanism will provide an application speed up.

Figure 6 and Figure 7 present the send-receive time for data packets. A send-receive communication allows the data copy time from the network and to the network to be incorporated in a manner that can be compared to other communication types. If extra data copy time is an issue within the framework, the result will be seen in the send-receive for the communication mechanism.

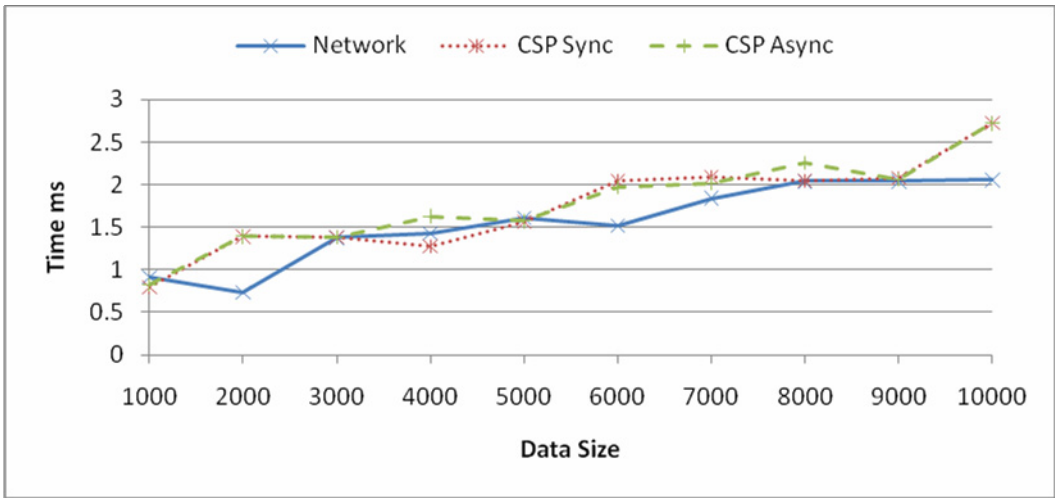

Figure 6: Send-Receive Times 1000 to 10000

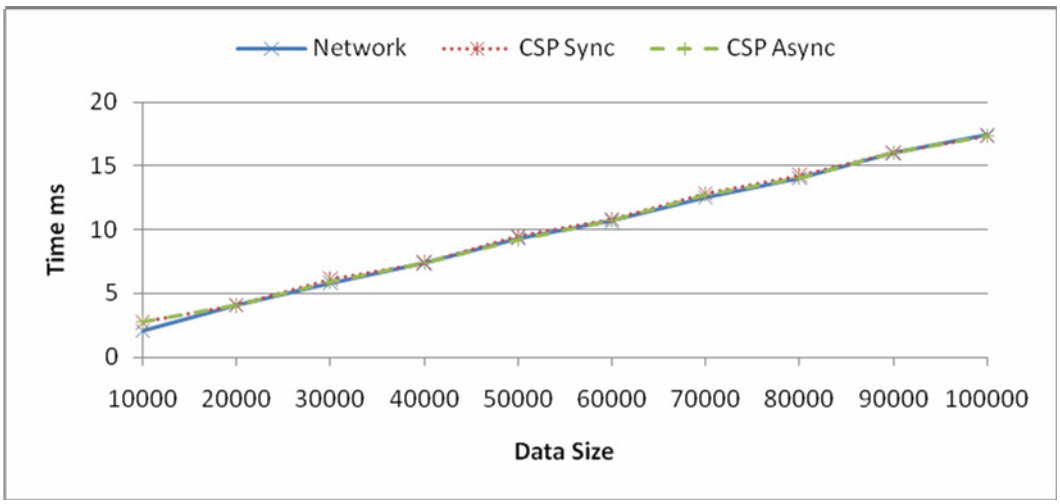

Figure 7: Send-Receive Times 10000 to 100000

Figure 6 and Figure 7 indicate that performance is almost identical for send-receive for basic network communication, synchronous networked CPA and asynchronous networked CPA. The reason that synchronous behaviour appears similar is due to the resulting data being sent back to the host immediately after the acknowledgement packet. By doing this, the overhead of the acknowledgement message becomes almost zero in comparison to the main overhead of sending data.

Figure 8 presents the actual throughput of the send-receive operation for the different scenarios. Unlike Figure 4, there is a more gradual trend before a tail of at 5.8×10^6 bytes per second (which is doubled to approximately 92.8 Mbit/s). The results in Figure 8 also initially show more noise, which is likely due to the small overheads of packets and occasional optimal packet sizes for sending.

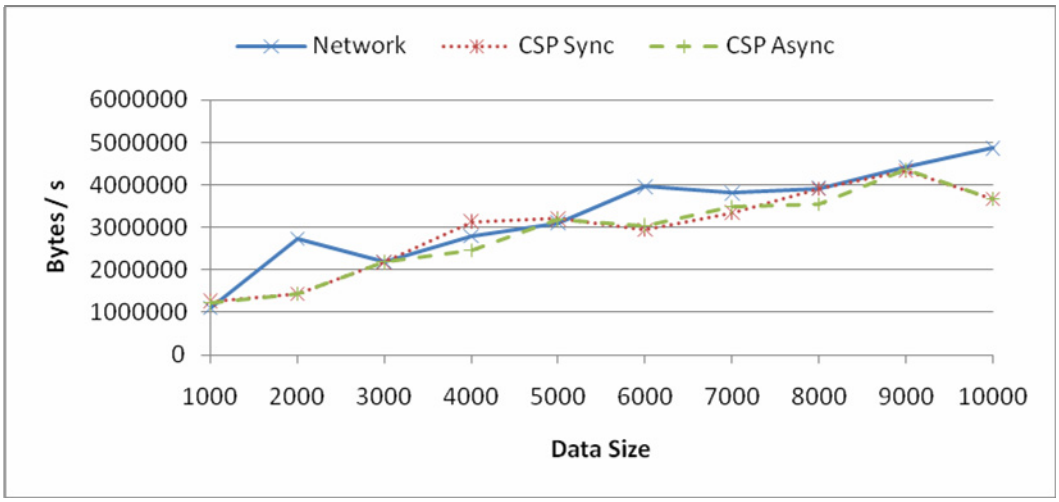

Figure 8: Send-Receive Throughput 1000 to 10000

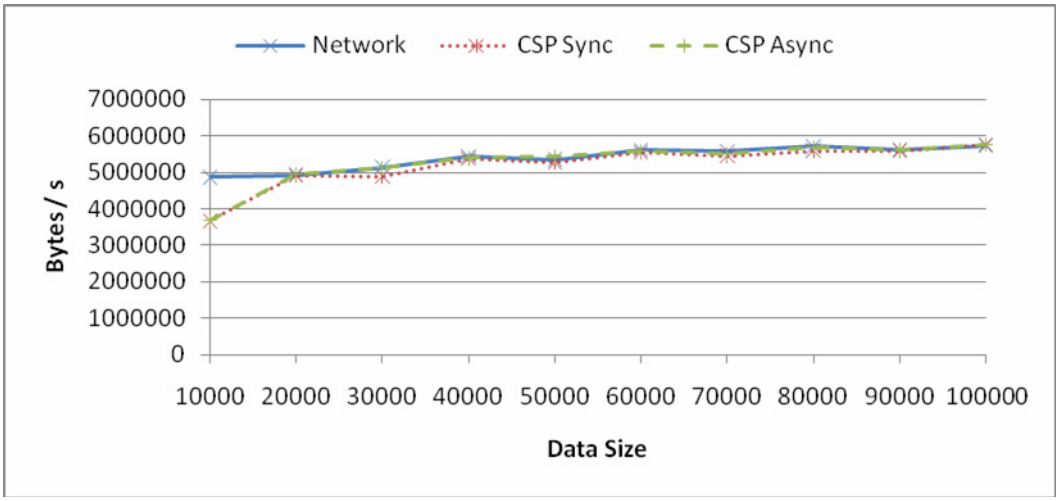

Figure 9: Send-Receive Throughput 10000 to 100000

4.2.3 Summary

The results presented comparing the distributed CPA architecture and protocol to standard network communications have shown that there is little overhead in using the CPA architecture as a distributed system communication layer. Asynchronous communication within the CPA architecture is almost identical to the baseline network, and synchronous communication has a small overhead due to the acknowledgment packet. Encouragingly, bandwidth is still very good for the distributed CPA architecture.

To give a better idea of how well the networked architecture performs in an actual parallel processing environment, the generation of Mandelbrot fractals was also undertaken. These results are presented in the next section. All results are taken from the mean of 100 runs of the test program.

4.3 Mandelbrot

For the Mandelbrot results two approaches were taken. For each test, bitmap images of 3500 × 2000 pixels were produced, each pixel representing a value in the Mandelbrot set. The first approach splits a single image processing into a number of segments (2 × 2, 4 × 4) and calculates the time taken to generate the complete image using a set of nodes. The second approach scales the Mandelbrot set so that a single 3500 × 2000 pixel image represents only part of the entire set.

4.3.1 Algorithm

The algorithm used to generate the individual data points of the Mandelbrot set is the typical escape time algorithm. The value generated for the escape time represents the grey scale colour value of an individual pixel of the final Mandelbrot fractal image. In C#, the algorithm is as follows:

```
float[,] Mandelbrot(int startX, int endX, int startY, int endY)
{
  float[,] results = new float[endX - startX, endY - startY];
  float xstep = 3.5f / (float)(endX - startX);
  float ystep = 2.0f / (float)(endY - startY);
  for (int x = startX; x < endX; ++x)
  {
    for (int y = startY; y < endY; ++y)
    {
      float x0 = -2.5f + ((float)x * xstep);
      float y0 = -1.0f + ((float)y * ystep);
      float x1 = 0.0f, y1 = 0.0f;
      int iteration = 0;
      int maxiteration = 1000;
      while ((x1 * x1) + (y1 * y1) <= (2.0f * 2.0f)
             && iteration < maxiteration)
      {
        float xtemp = (x1 * x1) - (y1 * y1) + x0;
        y1 = 2 * x1 * y1 + y0;
        x1 = xtemp;
        ++iteration;
      }
      if (iteration == maxiteration)
        results[x, y] = 0.0f;
      else
        results[x, y] = 1.0f / (float)iteration;
    }
  }
  return results;
}
```

The parameters `startX`, `endX`, `startY` and `endY` specify the rectangular region of the set to compute. Arrays of float data form the basis of the result packet to avoid the serialisation overhead that would occur from serialisation of non-basic types.

4.3.2 Sequential Mandelbrot Results

As a baseline, a sequential execution of the given algorithm was performed. This provided an execution time of 15242 ms for the generation of a 3500 × 2000 bitmap image of the Mandelbrot set when ignoring cache effects. This value should therefore be the approximate average time taken to generate one segment of the scaled version of the Mandelbrot set.

4.3.3 Parallel Mandelbrot Architecture

A standard farmer pattern has been used to perform the parallelisation. A central server produces the work, and *n* clients perform the work. As the machines being used in the experimental framework are dual core, each client runs two Mandelbrot Worker processes in parallel. A Buffer process exists on each client that requests work from the Mandelbrot Producer process, ensuring that a local packet of data is always available for processing by the Mandelbrot Workers whenever they request a new data packet. Figure 10 illustrates the overall system architecture.

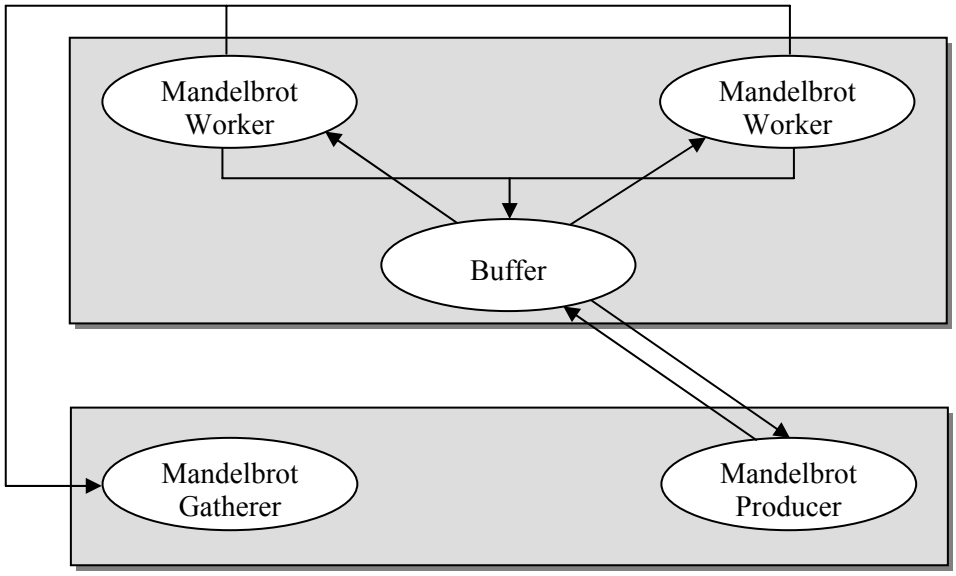

Figure 10: Mandelbrot Architecture

For all parallel Mandelbrot, the set is split using a tiled approach rather than optimising for the architecture used. Although this approach will not provide the fastest resolution of the set, it provides a method that can be scaled trivially.

4.3.4 Parallel Mandelbrot Results

To estimate the amount of time taken to actually transfer data from one of the workers to the server, the sequential test was run again using the parallel architecture. A single client and the server were run on the same machine (Local Distributed) and then the client and server were run on separate machines. Figure 11 presents the results for these two tests against the sequential algorithm time.

Figure 11 illustrates that running the client and server on the same machine leads to an approximate 4000 ms increase in processing time over the sequential time (the Local Distributed time is 19469 ms). This result is poor as on a dual core machine, the client and server should be effectively running on separate cores, thereby parallelising the execution

to a certain degree. There is data transfer of 2.8×10^7 bytes from the client to the server, but over a local host connection transfer should have been reasonable.

The application having the client and server operating on different machines provides similar performance to a Local Distributed system, the time for this experiment being 21590 ms. From the data gathered for these two tests, it can be determined that there is an approximate 5000 ms overhead for returning a single segment of data from a client to the server. Therefore, we have an approximate bandwidth of 45 Mbit/s.

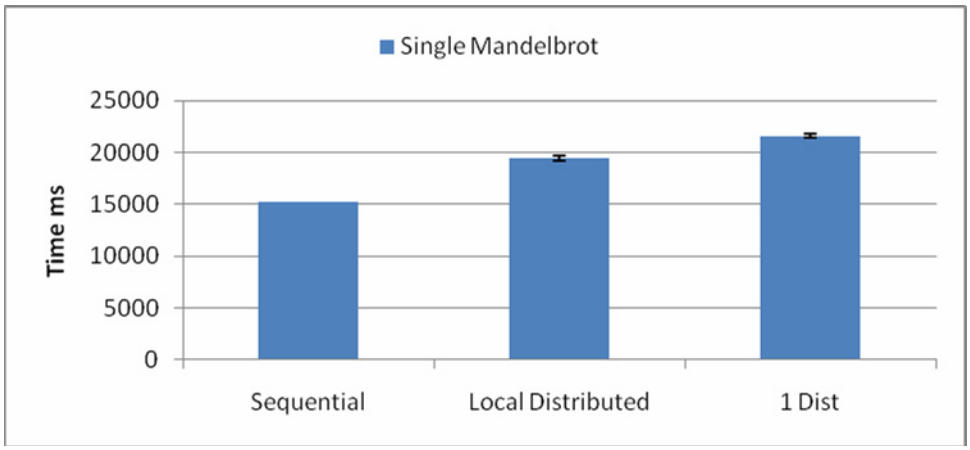

Figure 11: Single Mandelbrot Piece

Figure 12 presents the results of splitting the single Mandelbrot set of 3500 × 2000 pixels into 4 and 16 segments, and processing the data across 2 Nodes (4 cores) and 4 Nodes (8 cores).

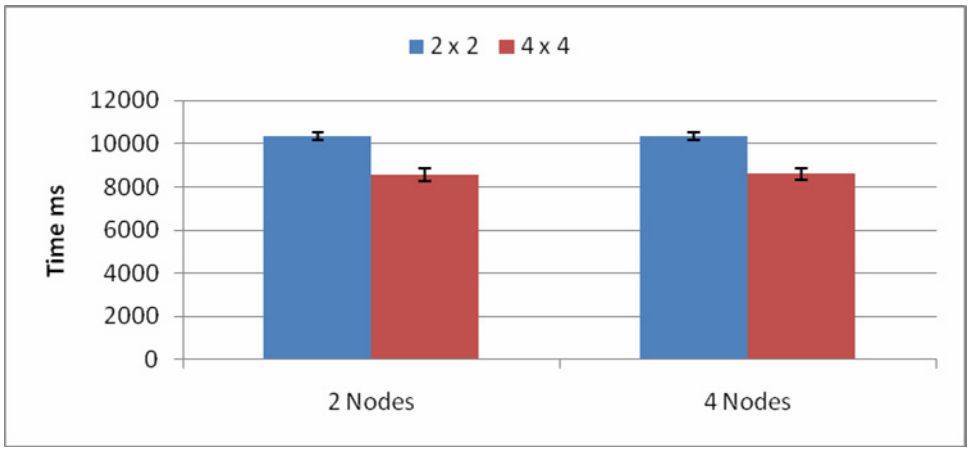

Figure 12: Single Mandelbrot Piece Segmented

For the 2 × 2 segmented set, performance with 4 cores and 8 cores is roughly the same. This is to be expected, as there are only 4 segments to process. However, the 4 × 4 segmented set also shows little speed up from running across 4 cores to 8 cores. It would be expected that some speedup was present, but it should be considered that the tiled approach to generating the Mandelbrot does not give optimum results.

In comparison to the standard sequential generation of the Mandelbrot set, we see a speedup from 15242 ms to 10343 ms when segmented into 2 × 2, and 8540 when segmented into 4 × 4. This can be considered a reasonable improvement when parallelising across the network.

Figure 13 presents the results for increasing the resolution of the data set using Local Distribution, 1 distributed machine, and 2 distributed machines. The 2 × Scale provides a

final resolution of 7000 × 4000 pixels, and the 3 × Scale a final resolution of 10500 × 6000 pixels. For all scaled results, the Mandelbrot set is still tiled, but with each tile having a 3500 × 2000 resolution.

Comparing the results taken from the client and server operating on the same machine to the client and server operating on different machines, we see a slight speed up for the 2 x Scale, and a greater speed up for the 3 × Scale Mandelbrot set. As the client is not contending with the server for processor time, this is to be expected. However, when the work is spread across two machines, there is little difference on the results when one machine does the work. We can therefore determine that there is no tangible speedup when adding more processors to the problem in this case. In this instance, we are seeing I/O bound behaviour, where the server is receiving too much data to process in a manner that provides parallel speedup.

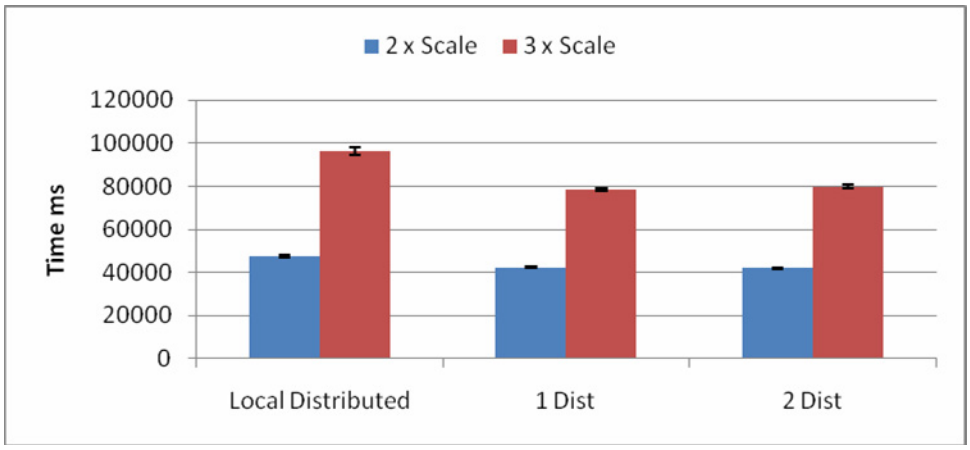

Figure 13: Scaled Mandelbrot Local, One and Two Nodes

Figure 14 indicates that the same behaviour continues when scaling up from 2 Nodes to 4 Nodes when we scale the resolution by 2, 3, and 4 for the Mandelbrot set.

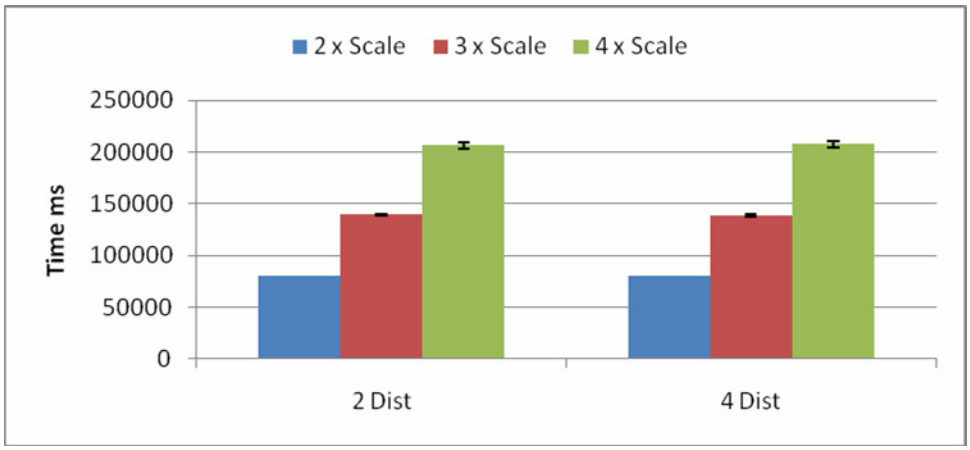

Figure 14: Scaled Mandelbrot Two and Four Nodes

Figure 15 presents the ongoing trend when using 4 Nodes (8 cores) to process progressively higher resolution results for the Mandelbrot set. When we look at this trend, we see that the general progression of time to process the data is similar to the scaling of the resolution. This is expected due to the scaling of the amount of work and the amount of data being transferred.

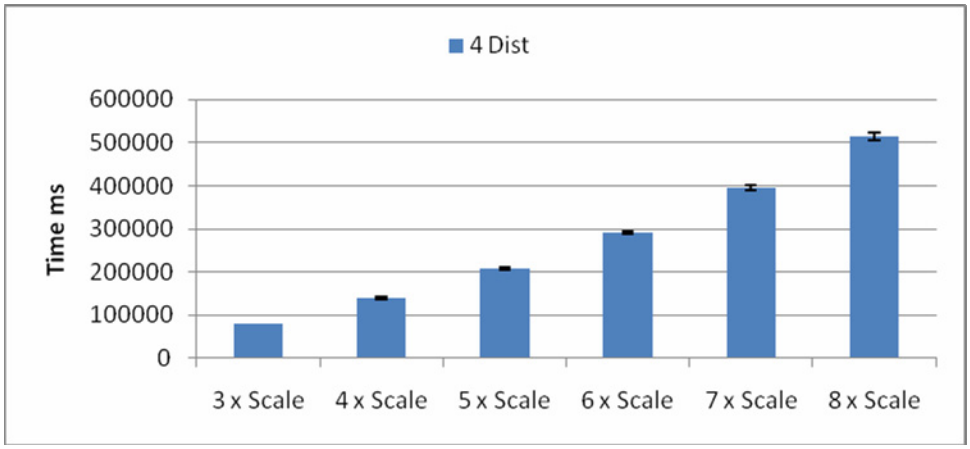

Figure 15: Scaled Mandelbrot Four Nodes

4.3.5 Data Throughput

To help analysis of the ongoing trend, Table 1 presents the number of data points and bytes being processed to 3 significant digits for the best performing configuration at the given scale. Looking at these results shows that the performance is actually increasing gradually even when scaling up the amount of work to perform.

Table 1: Mandelbrot Data Throughput

Scale	Data Points	Bytes	DP / s	Bytes / s
1 × Scale	7×10^6	2.8×10^7	3.25×10^5	1.3×10^5
2 × Scale	2.8×10^7	1.12×10^8	6.67×10^5	2.66×10^6
3 × Scale	6.3×10^7	2.52×10^8	8.04×10^5	3.21×10^6
4 × Scale	1.12×10^8	4.48×10^8	8.04×10^5	3.22×10^6
5 × Scale	1.75×10^8	7×10^8	8.46×10^5	3.38×10^6
6 × Scale	2.52×10^8	1.008×10^9	8.65×10^5	3.46×10^6
7 × Scale	3.43×10^8	1.372×10^9	8.69×10^5	3.47×10^6
8 × Scale	4.48×10^8	1.792×10^9	8.71×10^5	3.48×10^6

4.3.6 Summary

The results presented in this section illustrate that there is very little if any general communication performance difference between the distributed CPA architecture and standard network communications. In particular, when the results compared here are compared to previous work using mobile devices [4], we can see that there is no device strain being caused due to large data packets being sent to a single device. This is to be expected, due to machines in use in these experiments being significantly more powerful than the mobile devices in previous work.

The Mandelbrot experimental results are less encouraging. Although there is a speedup when breaking down a problem into smaller pieces, there appears to be no speedup when adding more processors to the parallel application. Further work is required on other problems to determine if this is a particular case of the Mandelbrot problem, or the distributed CPA framework itself when dealing with multiple nodes.

5. Conclusion and Future Work

The work presented here has shown that the distributed CPA architecture and protocol have good performance when compared to standard network communication. Although the Mandelbrot experiments have not provided a figure showing a significant speedup when adding more processors, further experimentation in this area may prove useful.

Currently, work is ongoing in implementing the protocol and architecture in other runtimes, with a particular emphasis at this point in occam-π. However, at this stage there is a major drawback when implementing the architecture in occam-π, due to the lack of data structure support. The distributed CPA architecture relies on a number of dynamically sized lookup tables internally to operate numerous inter-node connections and virtual channels. Currently, this can only be achieved in occam-π using arrays, which do not allow the level of complexity of data structure required. Although a primitive communication between JCSP and occam-π has been achieved, the system is at best unstable.

Therefore, a current investigation into using C++CSP is underway to see if it can be used to create a native library that occam-π, JCSP, etc. can hook into. Although a possible avenue to avoid the lack of data structure issue in occam-π, this would require running a system level service which is hooked into by the relative framework. This approach could cause conflicts with scheduling between the service and the runtime framework, which will need further investigation.

The main outcome of this work is that there is now inter-framework communication between two separate CPA inspired libraries using the new protocol and architecture for distributed CPA communication, and the performance of the two libraries are the same when communicating across a network. With this capability, and the general good performance of the architecture and library, it can be argued that the distributed CPA architecture and protocol are highly suited for any distributed system application where CPA behaviour is required.

Acknowledgements

The author would like to acknowledge the work of Julien Mateos for his help in building and testing the new version of CSP for .NET.

References

Error! Bookmark not defined.

[1] P.H. Welch, "Process Oriented Design for Java: Concurrency for All," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000)*, 2000, pp. 51-57.

[2] C. A. R. Hoare, *Communicating Sequential Processes*.: Prentice Hall, Inc., 1985.

[3] P. H. Welch, N. C. C. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh, "Integrating and Extending JCSP," in *Communicating Process Architectures 2007*, 2007, pp. 349-370.

[4] Kevin Chalmers, Jon Kerridge, and Imed Romdhani, "A Critique of JCSP Networking," in *Communicating Process Architectures 2008*, 2008.

[5] Kevin Chalmers and Sarah Clayton, "CSP for.NET Based on JCSP," in *Communicating Process Architectures 2006*, Edinburgh, 2006, pp. 31-41.

[6] P. H. Welch, J. R. Aldous, and J. Foster, "CSP Networking for Java (JCSP.net)," in *International Conference of Computational Science - ICCS 2002*, 2002, pp. 695-708.

[7] J. M. Bjørndalen, B. Vinter, and O. Anshus, "PyCSP - Communicating Sequential Processes for Python," in *Communicating Process Architectures 2007*, 2007, pp. 229-248.

[8] N. C. C. Brown and P.H. Welch, "An Introduction to the Kent C++CSP Library," in *Communicating Process Architectures 2003*, 2003, pp. 139-156.

[9] N. C. C. Brown, "C++CSP2: A Many-to-Many Threading Model for Multicore Architectures," in *Communicating Process Architectures 2007*, 2007, pp. 183-205.

[10] Inmos Limited, "The T9000 Transputer Instruction Set Manual," SGS-Thompson Microelectronics, 1993.

[11] M. Schweigler and A. T. Sampson, "pony - The occam-pi Network Environment," in *Communicating Process Architectures 2006*, 2006, pp. 77-108.

[12] J. M. Bjørndalen and A. T. Sampson, "Process-Oriented Collective Operations," in *Communicating Process Architectures 2008*, 2008.

[13] N. C. C. Brown, "C++CSP Networked," in *Communicating Process Architectures 2004*, 2004, pp. 185-200.