Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Scaling PyCSP

Rune Møllegaard FRIBORG [a,1], John Markus BJØRNDALEN [b] and Brian VINTER [c]
[a] *BiRC, University of Aarhus, Denmark*
[b] *Department of Computer Science, University of Tromsø, Norway*
[c] *Niels Bohr Institute, University of Copenhagen, Denmark*

Abstract. PyCSP is intended to help scientists develop correct, maintainable and portable code for emerging architectures. The library uses concepts from Communicating Sequential Processes (CSP) and is implemented in the Python programming language. This paper introduces a new channel implementation and new process types for PyCSP that are intended to simplify writing programs for clusters. The new processes and channels are investigated by running three benchmarks on a cluster, using up to 512 CPU cores. The results show that PyCSP scales well, and that the introduction of a mobile channel home improves the performance of all three benchmarks.

Keywords. PyCSP, CSP, clusters, channels, processes

Introduction

Maintaining scientific code is a well-known challenge; many applications are written by scientists without any formal computer science or software engineering qualifications, and are usually grown ”organically” from a small kernel to hundreds of thousands of lines of code.

These applications have traditionally targeted simple single core systems and have still grown to a complexity where the cost of maintaining the codes is a limiting factor, and where the continued correctness of the code is often questionable.

PyCSP [1] is intended to help scientists develop correct, maintainable and portable code for emerging architectures. Python is highly suited for scientific applications. While it is interpreted and thus very slow, scientific libraries efficiently utilize the underlying hardware. CSP provides a formal and verifiable approach to controlling concurrency [2], fits directly into scientific workflows, and maps directly onto many graphical tools that present scientific workflows such as Taverna [3], Knime [4] and LabView [5].

The intended users for PyCSP are not computer scientists, but scientists in general. General scientists can not be expected to learn CSP as formulated by Hoare, thus the approach to controlling concurrency is based on CSP, but does not require any knowledge in CSP. The key elements of controlling concurrency using PyCSP are presented in Section 3.

CPU’s are produced with multiple cores today and every announced future CPU generation [6] seems to feature an ever increasing number of cores. As single core performance increases very slowly, researchers are required to exploit this parallel hardware for increased performance. To this end a number of parallel libraries like BLAS and programming tools like Intel Parallel Studio [7] are appearing. Unfortunately parallel libraries are often not enough to achieve acceptable speed and even with advanced tools parallel programming remains a source of added complexity and new bugs in software development.

This paper investigates using PyCSP for running scientific applications on a cluster with up to 512 GPU cores. The cluster is comprised of several computers, each with multi-core

[1]Corresponding Author: *Rune Møllegaard Friborg,* E-mail: `runef@birc.au.dk`.

CPU's. Thus, there are several layers of concurrency to control and coordinate. For this reason, PyCSP has been updated with a channel supporting distributed communication and a new process type supporting remote execution.

1. Related Work

During the last decade we have seen numerous new libraries and compilers for CSP. Several implementations are optimized for multi-core CPU's that are becoming the de-facto standard when buying even small desktop computers. occam-π [8], C++CSP [9] and JCSP [10] are three robust CSP implementations of CSP. C++CSP and JCSP are libraries for C++ and Java, while occam-π uses CSP inherently in the programming language.

Programming clusters of multi-cores is at best a nontrivial exercise, clusters may be programmed using MPI [11] or modern PGAS languages like UPC [12] for a C-style language, distNumPy [13] for Python style programming or even Fortress, X10, or Chapel [14] if the programmer is willing to learn a completely new programming language. These languages all suffer from the fact that they are fundamentally parallel programming languages, not concurrent languages. Google's language [15] Go. is concurrent but channels are not supported directly across the network so it does not scale seamlessly.

Two other CSP libraries have previously introduced cluster computing capabilities. occam-π through PONY [16] and JCSP [17]. The presented PyCSP aims for complete transparency using a single channel model, while both occam-π and JCSP requires the explicit creation of networked channels or channel ends. If all channels in occam-π is created with network-enabled shared channel ends, then it may be comparable to the presented PyCSP.

2. CSP

The Communicating Sequential Processes algebra, CSP [2], was introduced more than 25 years ago and while it was highly popular and thoroughly investigated in its first years, interest dropped off in the late 1980s because the algebra appeared to be a solution in search of a problem, namely modeling massively concurrent processes and providing tools to solve many of the common problems associated with writing parallel and concurrent applications.

CSP provides many attractive features with respect to the next generation processors; it is a formal algebra with automated tools to help prove correctness, it works with entirely isolated process spaces, thus the inherent coherence problem is eliminated by design, and it lends itself to being modeled through both programming languages and graphical design tools.

3. PyCSP

PyCSP provides an API that can be used to write concurrent applications based on CSP. PyCSP was introduced in 2007 [1] and has since been revised in 2009 [18] and in 2012 based on a distributed channel design [19]. PyCSP is now available in two separate versions: Parallel and greenlets. Both are packaged in single modules to motivate the developer to switch between them as needed. A common API is used for both implementations making it trivial to switch between them: compositional elements can be written once and used for both implementations. When switching from parallel to greenlets, the PyCSP application may execute differently as processes may be scheduled in another order and less fairly. Hidden latencies may also become more apparent when all other processes are waiting to be scheduled. Having several implementations sharing one PyCSP API was presented in [20]. The pycsp.greenlets implementation is for fine-grained CSP networks running on a single core and pycsp.parallel is for coarse-grained CSP networks running on multi-core, many-core and distributed architectures.

The two implementations:

- `pycsp.greenlets` - This implementation uses co-routines instead of threads. Greenlets [21] is a simple co-routine implementation available as a Python module. It provides the possibility of creating millions of CSP processes in a single CSP network. This version is optimal for single-core architectures since it provides the fastest communication, but with no parallelism.
- `pycsp.parallel` - All channels are network enabled and support communication with remote and local processes. This is the default implementation for PyCSP programs. Currently four different process types exists for this version.
  - `@process` - A CSP process implemented as an OS thread. The internal synchronization is handled by thread-locking mechanisms. Because of the Python Global Interpreter Lock[1], this is best suited for applications that spend most of their time in external routines that release the GIL.
  - `@multiprocess` - A CSP process implemented as an OS process. It is built on top of the multiprocessing module available in Python 2.6 and newer. This implementation is not affected by the Global Interpreter Lock, but has some limitations on a Windows OS, primarily caused by the absence of a Windows fork() call.
  - `@sshprocess` - A CSP process implemented as an OS process running on a remote host. The execution of the process is handled using the SSH protocol, thus this process requires a set of extra parameters to initiate the execution.
  - `@clusterprocess` - A CSP process implemented as an OS process running on a remote host listed in a node file. The remote host is selected based on a distribution scheme provided to the clusterprocess. The process implementation is suited for distributing large scale CSP networks on clusters and is explained in more detail in Section 3.2.

In this paper we focus exclusively on the pycsp.parallel implementation and particularly on the aspects facilitating the distribution of a single CSP network across cores and hardware nodes. These next sections explain the different aspects of the pycsp.parallel implementation.

3.1. Channels

All channels are network-enabled, untyped, any-2-any, can be buffered, and support input and output guards. The primary motivation behind the design of this channel is that scientific users of the PyCSP library should not have to think about whether they might be sending a channel end to a process that might be running in a remote location, or how they avoid invoking an output guard on a channel that does not support it. One of the powerful characteristics of CSP is that every process is isolated, which means that we can move it anywhere and as long as the channels still work, the process will execute the same.

Every Python interpreter importing pycsp.parallel will create one dispatch thread listening on one port and that port receives all the communication to channels and processes started in this Python interpreter.

When a new channel is created, it registers the dispatch thread at the local Python interpreter as the channel home. Processes can request channel ends for registered channels at the

[1]Python uses a Global Interpreter Lock, the GIL, to protect the interpreter when multiple threads execute Python code. The GIL limits concurrency when executing Python code, but libraries commonly mitigate the problem by releasing the GIL when executing external code.

```
# Python interpreter 1
A = pycsp.Channel("A") # Hosting channel A
print(A.address)       # Outputs ('192.168.1.16', 56562)

@pycsp.multiprocess
def Hello(cout, id)
  cout("Hello from Python interpreter 2")

# The channel location is passed to the other Python interpreter
# inside the channel end returned by A.writer()
pycsp.Spawn(Hello(A.writer()))
cin = A.reader()
print(cin())
```

Listing 1. Connecting channels implicitly.

```
# Python interpreter 1
A = pycsp.Channel("A") # Hosting channel A
print(A.address)       # Outputs ('192.168.1.16', 56562)
cin = A.reader()       # Request reading end
print(cin())

# Python interpreter 2
A = pycsp.Channel("A", connect=('192.168.1.16', 56562))
cout = A.writer()      # Request writing end
cout("Hello from Python interpreter 2")
```

Listing 2. Connecting channels explicitly.

dispatch thread. For a process to perform any action on a channel, a request must be posted to the channel home through the dispatch thread.

Each process registers a remote locking object at its local dispatch thread. Whenever the channel home tries to match a read request (process A) with a write request (process B), it does this in a critical region by acquiring locks at the remote locking objects of the involved processes A and B. This blocks all other channel homes from accepting requests owned by either process A or B. See [19] for a thorough description of the distributed channel model.

To connect to a hosted channel the location must be known, as there is no name server available for channels. The location of a channel can be fetched directly from the channel object and used implicitly by new processes (Listing 1), or channel locations can be specified explicitly (Listing 2). A channel end contains the address of the channel host and will automatically reconnect to the channel host if it is passed to a new Python interpreter.

The channel home is hosted where it is created, but our goal is that a simple application, such as the example in Listing 3, can be distributed on a set of nodes, simply by changing the process types. Unfortunately, when all channels are created on the same node and sent to processes spawned at other nodes, that node will be a bottleneck as all the channels will have that node as the home node. For this reason, we have made channel homes mobile: a process can request a channel to move its home to be hosted at another Python interpreter. A process may now acquire the home of a channel by invoking `ChannelEnd.become_home()` (Listing 4). This results in a more distributed communication pattern, which is able to scale with the number of processes (shown in Section 5).

```
@pycsp.process
def elementP(this_read, next_write):
  # Loop until poisoned
  while True:
    token = this_read()
    next_write(token + 1)

@pycsp.process
def connectionP(value, chan_out, chan_result):
  chan_out(value)
  print(chan_result())

  # Poison element processes
  pycsp.poison(chan_out)

if __name__ == "__main__":
  head = pycsp.Channel()
  current = head
  L = []
  # Create ring processes
  for i in xrange(N):
    next = pycsp.Channel()
    L.append(elementP(current.reader(), next.writer()))
    current = next

  # Start ring processes
  pycsp.Spawn(connectionP(0, head.writer(), current.reader()))
  pycsp.Parallel(L)

  # Wait for spawned processes and hosted channels to quit
  pycsp.shutdown()
```

Listing 3. Ring example.

When a process invokes `ChannelEnd.become_home()` on a channel that is hosted elsewhere, a new channel home is created locally and a MOVE request is posted to the active channel home. The old home temporarily suspends the channel and sends any buffered messages the channel may contain to the new location. It then notifies all the posted requests that the channel has been moved to the new address. After this, every process with a posted request updates the new address of the channel object and repost the request at the new location. The old home retains the MOVE notice to forward any requests until the PyCSP application quits.

The order of posted requests may easily be re-ordered during a call to `become_home`, thus priority and fairness can not be guaranteed during this operation. Also, if a process invokes `ChannelEnd.become_home()` on a channel that has already moved, the process is only updated with the location of the new channel and does not become the home of the channel.

3.2. Cluster Processes

To simplify cluster programming with PyCSP, a new process type has been introduced (Listing 5). The new type enables executing processes transparently on cluster nodes using a `@clusterprocess` decorator. The `@clusterprocess` decorator provides a similar environment to the decorated function as the `@process` (thread) and the `@multiprocess` decorators. Thus, to make a process with a computational workload run on a set of remote hosts, the only

```
@pycsp.clusterprocess
def elementP(this_read, next_write):

  # Move channel home to this Python interpreter
  this_read.become_home()

  # Loop until poisoned
  while True:
    token = this_read()
    next_write(token + 1)
```

Listing 4. Distributed element process from ring example in Listing 3.

```
@pycsp.clusterprocess(
  cluster_nodefile="$PBS_NODEFILE",     # Var or path to node file
  cluster_pin=None,                     # Index of host in node file
  cluster_hint='blocked', or 'strided' # Distribution scheme
  cluster_ssh_port=22,                  # Port to ssh server
  cluster_python='python'               # Python executable
)
def Remote_process(channel ends and function arguments):
  # process content
```

Listing 5. Specifying node distribution for cluster processes.

necessary change is to swap the `@process` decorator with the `@clusterprocess` decorator and provide a list of host names.

To start a Python interpreter on a remote host, we use the Python paramiko module[2], which implements the SSH protocol [22]. SSH was selected, since it is commonly used for logins on Linux clusters. The Python paramiko module and SSH servers are cross-platform and are also available for many other operating systems, such as Microsoft Windows and Mac OS X.

When the first clusterprocess is started, a node runner thread is initiated which handles each SSH connection. Only one SSH connection is made per cluster node and each executes the `pycsp.parallel.server` module in Listing 6. For the clusterprocess to work, it is assumed that all files are identical on all cluster nodes. This is a practical requirement, as Python functions and modules can not be transferred properly using serialisation, but must be loaded directly from source in lines 26-33 (Listing 6). Communication to `pycsp.parallel.server` is handled using PyCSP channels and not SSH transport protocols. Using PyCSP channels allows for a more concurrent design, where it is the task of the `RunFunc` process (lines 9-13) to send the return value of the cluster process directly to the process initiating `Parallel`, `Spawn` or `Sequence`. The node file information is passed along to new clusterprocesses together with the current state of distribution. This ensures, that a self-spawning chain will use the desired distribution scheme and not put all the new clusterprocesses on the first entry in the node file. For a self-spawning process tree, the distribution scheme will not be correct, as the state of distribution at parallel processes are not known. In such a case, the user must alternate between using 'strided' and 'blocked' distribution schemes.

[2]https://pypi.python.org/pypi/paramiko

```
cwd, ip, port, input_name = sys.argv[1:] # Parameters
os.chdir(cwd)                            # Change to working dir

# Connect to channel at parent and request input end
input_chan = Channel(input_name, connect=(ip, int(port)))
input_chan_end = input_chan.reader()

# (Spawn) PyCSP MultiProcess
def RunFunc(output_chan_name, fn, args, kwargs):
    output_chan = Channel(output_chan_name, connect=(ip, int(port)))
    send_return_value = output_chan.writer()
    val = Parallel(Process(fn, *args, **kwargs))
    send_return_value(val[0])

try:
    while True:
        # Get next process
        val = input_chan_end()
        h, p, output_chan_name, scriptPath, funcName,
          args, kwargs, available_nodes = val

        # Update cluster node information
        nodefile, group_state = available_nodes
        NodePlacement().set_nodegroup(nodefile, group_state)

        # Load script
        sys.path.insert(0, os.path.dirname(scriptPath))
        moduleName = os.path.basename(scriptPath)
        if moduleName[-3:] == '.py':
            moduleName = moduleName[:-3]
        m = __import__(moduleName)
        # Retrieve function object from imported script
        fn = getattr(m, funcName)().fn

        # Spawn a runner
        pycsp.Spawn(
          MultiProcess(RunFunc, output_chan_name, fn, args, kwargs),
          pycsp_host=h, pycsp_port=p)

except ChannelPoisonException:
    pass # shutdown nicely
```

Listing 6. python -m pycsp.parallel.server.

4. Experimental Platform

The genomedk cluster in Aarhus is comprised of 96 nodes. Each node has two Intel Xeon E5-2670 CPU's @ 2.67 GHz (total of 16 cores per node) and 64GB memory @ 1600 MHz. The nodes are interconnected by 10 GigE and 1 GigE NIC's. For the benchmarks in this paper, a maximum of 32 nodes were used. The cluster runs Centos 6.3 and is primarily used for bioinformatics tasks.

```
@clusterprocess
def find_kNN(body_in, body_out, result_out, locals, k):
  best = {}
  for local_id, _ in locals:
    best[local_id] = []

  if USING_BECOME_HOME:
    body_in.become_home()

  body_out(locals)
  while True:
    bodies = body_in()

    if locals[0][0] == bodies[0][0]:
      result_out(best)
      return

    for local_id, local_params in locals:
      B = best[local_id]
      for body_id, body_params in bodies:

        dist = numpy.sum((body_params-local_params)**2)
        dist = math.sqrt(dist)
        B.append((dist,body_id))

      # Sorts on first element.
      B.sort()
      best[local_id] = B[:k]

    body_out(bodies)
```

Listing 7. find_kNN process.

5. Experiments and Results

To investigate the performance of a distributed PyCSP network we have executed three different benchmarks: a kNN experiment to perform a k nearest neighbour search, a token ring experiment to show the scaling of communication on independent channels and finally a throughput experiment to test the maximum obtainable bandwidth on a 10GigE link for different message sizes. All runs were executed two times and the measured performance results did not deviate by more than 2% between the two runs. The average of the two runs was used for the benchmark plots.

Also, to demonstrate the effect of the new mobile channel home, all three experiments compare two executions with and without the use of the mobile channel home.

In the kNN experiment we run one `find_kNN` process per core (Listing 7). The input data set is split into smaller sets and divided among the `find_kNN` worker processes all connected in a ring. Every worker process is given a local set which it will pass around the ring. In each pass the worker execute a kNN algorithm on the local set and the received set, until finally all results are collected through a result channel and joined to produce the end result. Two different data input sizes have been selected, to test different computation to communication ratios. The experiment took 867 seconds for 204800 data elements on 512 cores and 266 seconds for 102400 data elements on 512 cores.

Figure 1 shows the results from running the kNN experiment on the cluster. To reduce the run time of the entire experiment, we did not run the experiment with fewer than 32

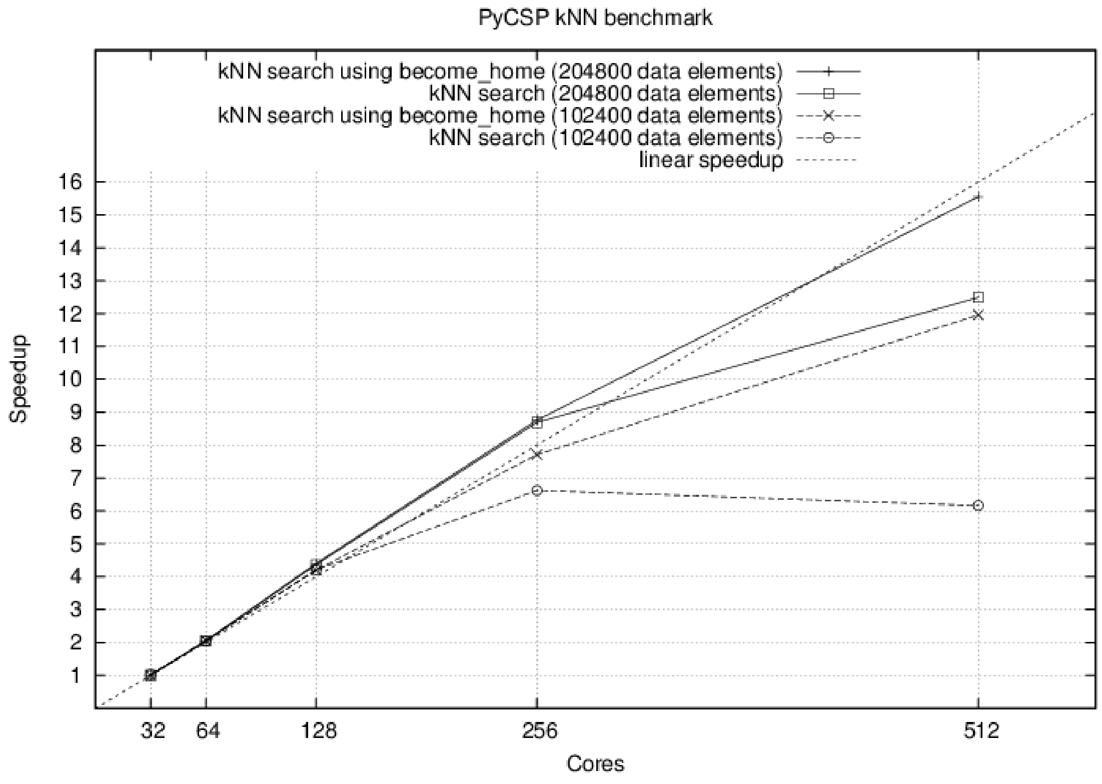

Figure 1. Running the kNN experiment. Speedup from 32 to 512 cores on the cluster. The runs using `become_home` moves the channel homes to the individual `find_kNN` processes.

cores. The results show that moving the channel homes to the individual `find_kNN` processes improves to performance considerably. The result for 204800 elements using `become_home()` show close to linear speedup for up to 512 cores. At 128 and 256 nodes, we observe a higher than linear speedup (super-linear speedup). The data elements fit better into the cache of the CPU's when split into smaller slices, which is the case for the input data size of 204800 elements split between 128 or 256 `find_kNN` processes.

Figure 2 shows the number of messages per second that are transmitted in a ring of PyCSP processes, similar to the one created in listing 3. The number of concurrent tokens were increased from 1 to 256. The linear scaling tells us that the channel homes have been successfully moved to the individually processes, thus eliminating a central bottleneck in the channel communication.

The throughput benchmark in Figure 3 shows the bytes/s transmitted from one 16-core node to another 16-core node on the cluster. The CSP network is a ring design, but instead of putting the first 16 processes on one node and the next 16 processes on another (blocked distribution scheme), the processes are distributed in a round-robin fashion (strided distribution scheme). This design makes every channel communication a communication from one node to another. The throughput on the 10GigE link has then been computed by running a ring network of size 32 with 16 concurrent tokens. For the local communication we ran the same experiment with multiprocessing instead of clusterprocesses. In both cases, the message size was varied from 1 byte to 100Mbyte, as this is the expected usage scenario. PyCSP supports larger messages than 100Mbyte, though the performance is lower for large messages.

The results show a throughput reaching 9.3Gbit on the 10GigE link when the message size is at 1Mbyte. Below 1Mbyte the latency of a channel communication results in a drop in throughput. Above 1Mbyte there is a overhead in handling the larger sizes for both local and 10GigE. This issue has been investigated through profiling of PyCSP and have shown that the time spent serialising Python data objects does not scale linearly with the size of data. Also,

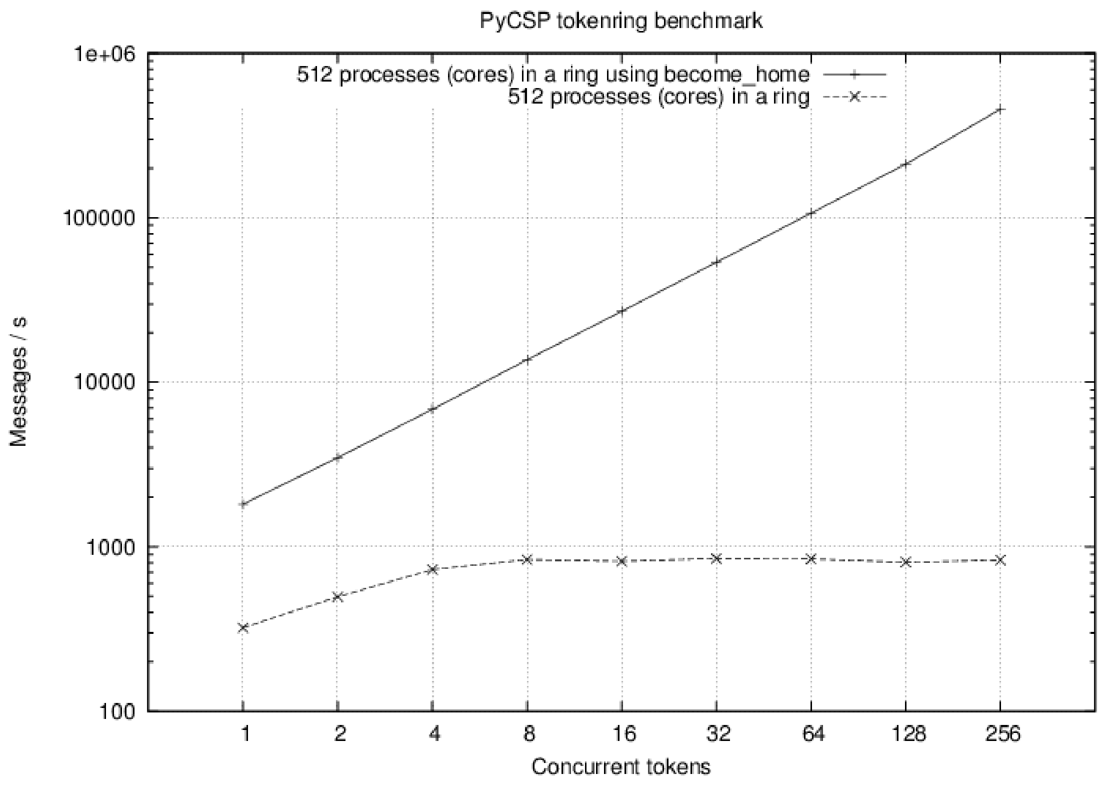

Figure 2. PyCSP token ring benchmark on the cluster.

there is a considerable drop in performance when the size of the data objects is above the available L3 cache on the CPU. Serialisation of Python data objects is a necessary step and is currently performed using the standard Python module pickle. During this serialisation, the data is written to memory and if the data can not fit in the cache of the CPU, the data must be sent to main memory before the operation can complete. A possible solution to this would be to make the serialisation step stream data directly to a socket stream. We are confident that this can be solved in the future.

From the results in Figures 2 and 3 we can also learn the following about the performance of a single dispatch thread hosting channels. It can communicate up to 850 messages per second when handling 512 channels for 512 cores in a distributed environment. It can reach a throughput of up to 550Mbit/s on a network link and up to 3.3Gbit/s on a local socket.

6. Conclusions

This paper introduces a new channel implementation and new process types for our Python CSP library, PyCSP, that are intended to simplify writing programs for clusters.

The new channel implementation adds the challenge of how to create a distributed channel network, without requiring the knowledge of each hostname and port number. This challenge is solved by making the channel home mobile, such that a process may request to become the home of a channel. Experiments in the paper shows that without the distribution of channel homes, PyCSP can not scale.

The new processes and channels are investigated by running three benchmarks on a cluster. The experiments show that PyCSP scales well for the application benchmark, with close to linear scaling for up to 512 CPU cores. The tokenring benchmark shows that the use of mobile channel homes have completely eliminated a communication bottleneck. For the throughput experiment, we are able to use nearly all available bandwidth with 1Mbyte messages, but the throughput quickly drops for larger messages. The serialisation of Python

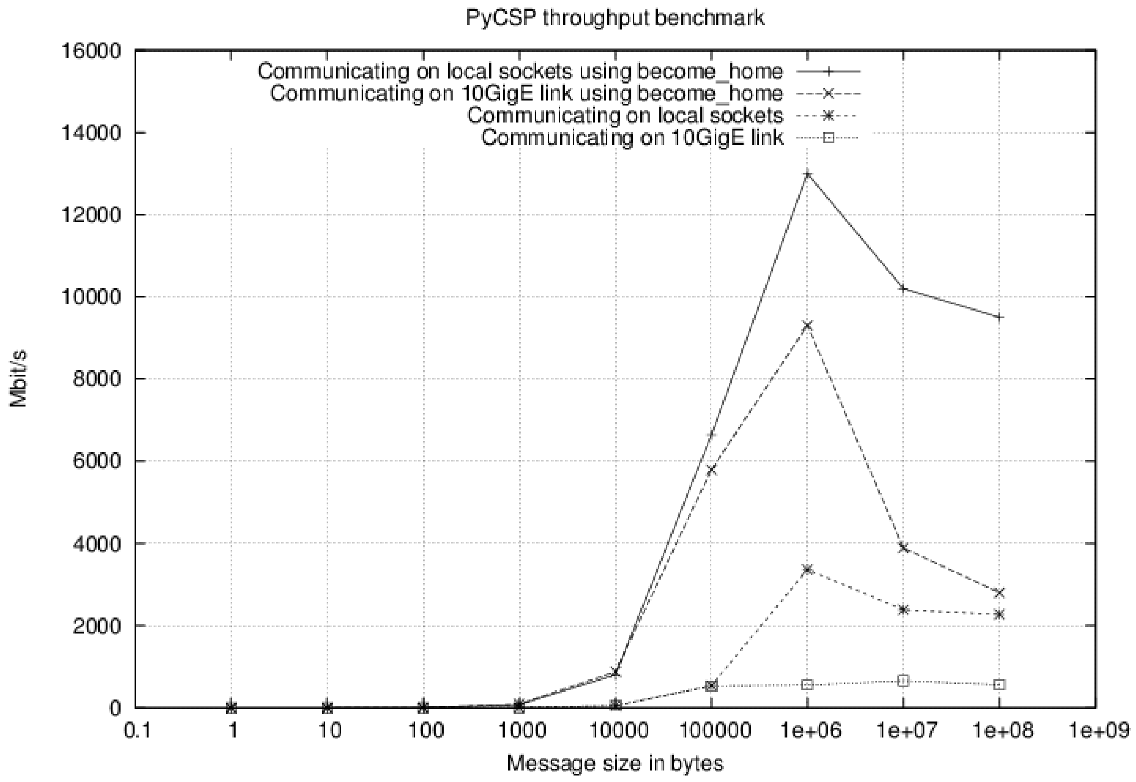

Figure 3. PyCSP throughput benchmark on the cluster.

objects is found to be the bottleneck and we expect that using streams between serialisation and sockets will solve the issue.

With these results PyCSP is now able to support scalable scientific applications on clusters. Scientific applications using PyCSP composed by other PyCSP networks. The kNN benchmark could be such a composed CSP network performing numerous massive kNN searches on-the-fly for other processes in a larger CSP network.

References

[1] John Markus Bjørndalen, Brian Vinter, and Otto Anshus. PyCSP - Communicating Sequential Processes for Python. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, pages 229–248. IOS Press, Netherlands, Jul 2007.

[2] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM, 21(8):666-677*, pages 666–677, August 1978.

[3] Taverna Project. http://taverna.sourceforge.net.

[4] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. Knime: The konstanz information miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.

[5] LabView. http://www.ni.com/labview/.

[6] Brian Vinter. Next Generation Processes. In B.H.V Topping and P. Ivanyi, editors, *Parallel, Distributed and Grid Computing for Engineering*, Computational Science, Engineering and Technology, pages 21–33. Saxo-Coburg Publications, 2009.

[7] Intel Parallel Studio. http://software.intel.com/en-us/intel-parallel-studio-home/.

[8] Peter H. Welch and Frederick R. M. Barnes. Communicating Mobile Processes - Introducing occam-π. In *Communicating Sequential Processes*, Lecture Notes in Computer Science, pages 175–210. Springer-Verlag, 2005.

[9] Neil C. Brown. C++CSP2: a Many-to-Many Threading. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–206. IOS Press, Netherlands, Jul 2007.

[10] Peter H. Welch, Neil C. Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369. IOS Press, Netherlands, Jul 2007.

[11] MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum*, March 1994.

[12] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[13] Mads Ruben Burgdorff Kristensen, Yili Zheng, and Brian Vinter. PGAS for Distributed Numerical Python Targeting Multi-core Clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 680–690. IEEE, 2012.

[14] Ewing Lusk and Katherine Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *Parallel Processing Letters*, 17(01):89–102, 2007.

[15] Robert Griesemer, Rob Pike, Ken Thompson, et al. The Go programming language, 2009.

[16] Mario Schweigler and Adam T. Sampson. PoNY – the occam-π Network Environment. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 77–108. IOS Press, Netherlands, Sep 2006.

[17] Peter H. Welch and Brian Vinter. Cluster Computing and JCSP Networking. In James S. Pascoe, Roger J. Loader, and Vaidy S. Sunderam, editors, *Communicating Process Architectures 2002*, pages 203–222. IOS Press, Netherlands, Sep 2002.

[18] Brian Vinter, John Markus Bjørndalen, and Rune M. Friborg. PyCSP Revisited. In *Communicating Process Architectures 2009*. IOS Press, Netherlands, Nov 2009.

[19] Rune M. Friborg and Brian Vinter. Verification of a Dynamic Channel Model using the SPIN Model Checker. In *Proceedings of WoTUG 33 / CPA 2011*. IOS Press, Netherlands, June 2011. ISBN 978-1-60750-773-4.

[20] Rune M. Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In *Communicating Process Architectures 2009*. IOS Press, Netherlands, Nov 2009.

[21] Greenlet: Lightweight in-process concurrent programming. http://pypi.python.org/pypi/greenlet.

[22] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) protocol architecture. 2006.