Cluster Computing and JCSP Networking

Brian Vinter
Department of Mathematics and Computer Science
University of Southern Denmark, Odense, Denmark
vinter@imada.sdu.dk

Peter H. Welch
Computing Laboratory, University of Kent at Canterbury, CT2 7NF, England
P.H.Welch@ukc.ac.uk

Abstract. Hoare's algebra of Communicating Sequential Processes (CSP) enables a view of systems as layered networks of concurrent components, generating and responding to events communicated to each other through channels, barriers and other (formally defined) synchronisation primitives. The resulting image and discipline is close to hardware design and correspondingly easy to visualise, reason about, compose and scale. JCSP is a library of Java packages providing an (occam) extended version of this model that may be used alongside, or as a replacement for, the very different threads-and-monitors concurrency mechanisms built into Java. The current release (JCSP 1.0) supports concurrency within a single Java Virtual Machine (which may be multi-processor). This paper reports early experiments with ***JCSP.net***, an extension of JCSP for the dynamic construction of CSP networks across distributed environments. The aims of ***JCSP.net*** are to simplify the construction and programming of dynamically distributed and parallel systems. It provides high-level support for CSP architectures, unifying concurrency logic within and between processors. The experiments are on some classical HPC problems, an area of work for which ***JCSP.net*** was not primarily designed. However, low overheads in the supporting infrastructure were a primary consideration – along with an intuitive and high-level distributed programming model (based on CSP). Results reported show JCSP holding up well against – and often exceeding – the performance obtained from existing tools such as *mpiJava* and IBM's *TSpaces*. The experimental platform was a cluster of 16 dual-processor PIII Linux machines. It is expected that future optimisations in the pipeline for the ***JCSP.net*** infrastructure will improve the results presented here. JCSP and ***JCSP.net*** were developed at the University of Kent.

1 Introduction

The aims of *JCSP.net*[1] are to provide simple ways to build efficient, richly functional, scalable, distributed and dynamically evolving systems. Its primitives are based on Hoare's algebra of Communicating Sequential Processes (CSP[2, 3, 4]), allowing applications to be formally and directly modelled. It enables the expression of concurrency at all levels of system – for example between parallel running devices (whether home, cluster or Internet distributed) or within a single machine (which may be multi-processor). The same concepts and theory are used regardless of the physical realisation of the concurrency – shared memory and distributed memory systems are modelled and programmed in the same way. This is especially useful for exploiting the mixed parallel architecture (CLUMPS) of our target platform, with no different programming techniques needed to deal with each style.

JCSP[5, 6, 7, 8, 9] views the world as layered networks of communicating processes, each layer itself being a process. Processes do not interact directly with other processes – only with CSP sychronisation objects (such as communication channels, event barriers, shared-memory CREW locks) to which groups of processes subscribe. In this way, race hazards are largely designed out and deadlock/livelock analysis simplified (and, if necessary, formalised in CSP for mechanically assisted verification). The strong de-coupling of the logic of each process from all others means we only have to consider one thing at a time – regardless of whether that *thing* is a process with a serial implementation or a network layer of concurrent sub-processes. A consequence is that CSP concurrency logic scales well with complexity (as well as distributing naturally across physically parallel systems). Such properties are not held by the built-in Java concurrency model (nor by parallel machine models such as MPI or Linda).

JCSP follows the CSP model pioneered by the occam[10, 11] concurrency language in the mid-1980s, which was the first commercial realisation of the theory that was both efficient and secure. It extends the occam model by way of some of the proposed extras (such as shared channels) for the never implemented occam3[12] language – and by taking advantage, carefully, of the dynamic features of Java.

The channel mechanism of CSP lends itself naturally to distributed memory computing – indeed, this was one of the most exciting capabilities of occam systems mapped to transputer networks. Currently, JCSP provides support for shared-memory multiprocessors and, of course, for uniprocessor concurrency.

This paper presents work with an early (alpha) release of *JCSP.net*, an extension of JCSP that extends the CSP concurrency model to networked systems. It enables the dynamic distribution and connection of JCSP processes with no central or pre-planned control (although that is easy to impose if necessary). Details of the underlying network fabric and control (such as multiplexing, flow-control, network addresses) are hidden from the JCSP programmer, who works entirely at the application level using CSP communication and synchronisation primitives.

A simple brokerage service – based on channel names (the Channel Name Server, CNS) – is provided to let distributed JCSP components find and connect to each other. Distributed JCSP networks may securely evolve, as components join, leave and migrate at run-time with no centralised or pre-planned control.

Processes themselves need not be aware that they are networked, which enables simple re-configuration and load balancing. The seamless unification of external and internal concurrency models, the dynamic construction of networked applications and the formal compositional basis of the underlying CSP model (which may either be ignored by users or employed directly for formal analysis and/or model checking) will be of interest, we hope, to workers in the field of HPC.

This paper assumes some familiarity with core JCSP mechanisms – channel interfaces, concrete channels (one-one, any-one etc.), synchronised communication, buffered plugins for channels, processes, parallel process constructors and alternation (i.e. the passive waiting for one from a set of events to happen).

The paper is structured as follows. Section 2 presents the basic model, semantics and API provided by *JCSP.net*, along with some generic templates for standard parallel decomposition paradigms (*farming* and *space division*). Section 3 describes the design aims for the *DANISH* benchmark suite. Section 4 outlines the alternative parallel mechanisms used for comparison (*mpiJava* and IBM's *TSpaces*). Section 5 presents the applications benchmarked in this work and Section 6 gives the results. Finally, some concluding remarks are made in Section 7.

2 Motivation

We want to use the same concurrency model regardless of the physical distribution of the processes of the system.

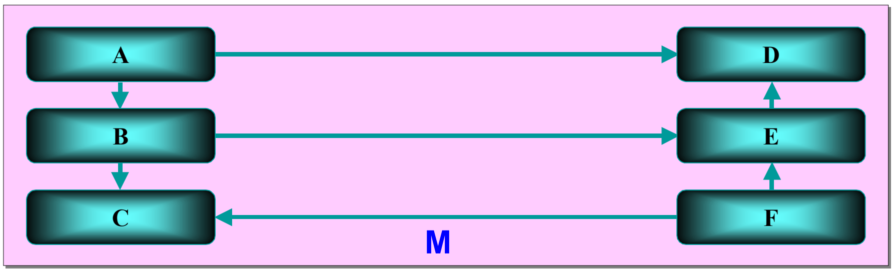

Figure 1(a). Single processor system

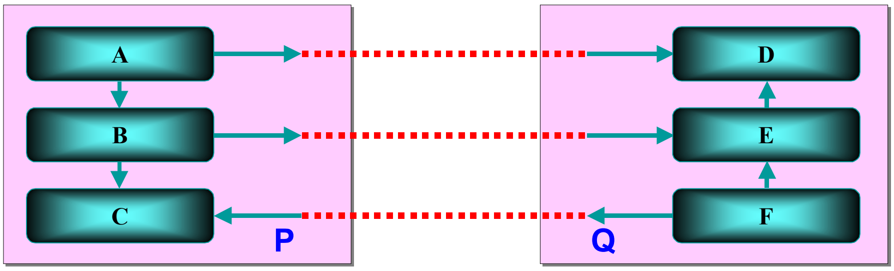

Figure 1(b). Two processor system

Figure 1(a) shows a six process system running on a single processor (**M**). Figure 1(b) shows the *same* system mapped to two machines (**P** and **Q**), with some of its channels stretched between them. The semantics of the two configurations should be the same – only performance characteristics may change.

2.1 Basic Infrastructure

JCSP.net provides mechanisms similar to the *Virtual Channel Processor* (VCP) of the T9000 *transputer* [13]. Users just *name* networked channels they wish to use. Details of how the connections are established, the type of network used, machine addresses, port numbers, the routing of application channels (e.g. by multiplexing on to a limited supply of socket connections) and the generation and processing of acknowledgement packets (to preserve synchronisation semantics for the application) are hidden. The *same* concurrency model is used for networked systems as for internal concurrency. Processes may be designed and implemented without caring whether the synchronisation primitives (channels, barriers etc.) on which they will operate are networked or local. This is as it should be.

The user does not need to know how, for example, Figure 1(b) is implemented. In fact, a bi-directional *link* is established between the two machines and the application-level channels are multiplexed over this – see Figure 2.

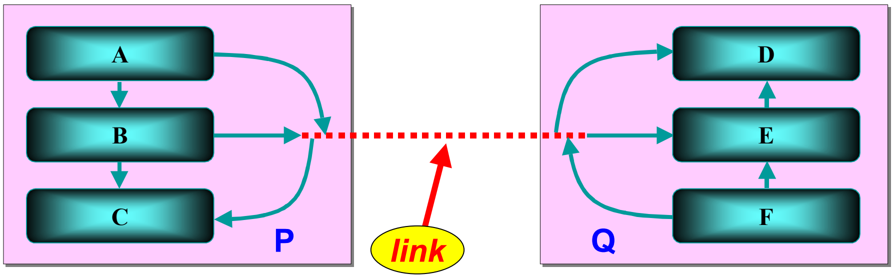

Figure 2. Channel multiplexing over a link

JCSP.net establishes one – and only one – such link between each pair of machines that contain processes wishing to communicate. Details of the underlying mechanisms (including a very simple software crossbar connecting application-level processes with link driver processes) may be found in [1].

2.2 Semantic Integrity

JCSP.net designers and programmers need no knowledge of the infrastructure outlined in the previous subsection. JCSP *processes* only see channel *interfaces* – for the most part, they need not even be aware whether the *actual* channels plugged into them are local or networked.

It is essential, however, that link multiplexing preserves the independence of the channels carried. So, the semantics of the networks in Figures 2 and 1(b) remain identical. For example, if process `E` is slow at taking (or simply refuses to take) input from its networked channel, this must not block communications from `A` to `D` nor from `F` to `C`.

Like normal ***JCSP*** channels, users may specify any amount (and several varieties) of buffering for networked channels – with zero buffering being the default. The link infrastructure introduces no additional buffering. Hence, the synchronisation semantics of Figures 1(b) and 1(a) are identical.

However, there are two *caveats*. Firstly, current *JCSP.net* network drivers use Java *serialisation* to transmit and receive objects – so objects for networked communication must implement the Java `serializable` interface.

Secondly, networked channel communication has *copy-semantics* – whereas across internal channels, communication is by *reference*. Fortunately, for *secure* JCSP designs (i.e. those free from shared-reference race-hazards), this will make no difference.

2.3 Establishing Network Channels

JCSP.net does not require applications to be pre-configured statically. For example, the number of participating machines or application processes need not be known in advance – nor the mapping of the latter to the former. The model is very dynamic – processes find each other by knowing (or finding out) the names (or locations) of the network channels they need to use. This is much more flexible than the static configuration offered, for example, by MPI.

At the lowest level, this mechanism is brokered by the *JCSP.net Channel Name Server* (CNS), which maintains a table of network addresses (e.g. IP-address/port-number, depending on the type of network), de-multiplexing keys (for navigating the software crossbar implementing the *JCSP.net* infrastructure on each node) and channel types (e.g. *streamed*, *overwriting*) against user-defined channel *names* (which can have a rich structure akin to URLs).

Within a processing node, which may itself be SMP, networked application channels are either *to-the-net* or *from-the-net*. These channels may have *one* or *any* number of application processes attached – which gives us the set {`One2NetChannel, Any2NetChannel, Net2OneChannel, Net2AnyChannel`}.

The '`2Net`' channels implement only `write` methods and the '`Net2`' channels only `reads`. In line with core JCSP[5, 6], only the '`2One`' channels may be used as *guards* in an `Alternative`.

To construct networked channels using the CNS, we only need to know their names. For example, at the reading end, the statement:

```
Net2OneChannel foo = new Net2OneChannel ("ukc.foo");
```

constructs the channel and necessary local multiplexing infrastructure, registering the name `"ukc.foo"` with the CNS (along with the network address of the constructing node and navigation information to enable correct de-multiplexing). If the CNS cannot be found or the name is already registered, an exception is thrown. Details of that supporting local infrastructure are reported in [1]. Note that this `foo` channel only has a `read` method.

At the writing end, the statement:

```
One2NetChannel foo = new One2NetChannel ("ukc.foo");
```

interrogates the CNS for registration details on `"ukc.foo"`, blocking until that registration has happened (time-outs may be set that throw exceptions), and constructs the channel together with its supporting infrastructure. *This* `foo` channel, which is in a different process and on a different machine to the previous one, only has a `write` method.

Each node maintains a table of machines to which it has open links. If a link does not exist to the machine registered as the *reading-end* of `"ukc.foo"`, that machine is contacted and the link is made (and the tables at each end updated).

If network channels are being constructed *sequentially* by an application, all input channels should be set up first – otherwise, there will be deadlock!

JCSP.net channels are *unidirectional* and support *any-one* communication across the network. This means that processes on any number of remote machines may open a named *output* network channel and use it safely – their messages being interleaved at the *input* end.

2.4 Networked Connections (Client-Server)

Establishing *two-way* communication, especially for a prolonged conversation between one of many remote clients and a common server, can be done with *channels* … but is not particularly simple or elegant. Individual network *reply* channels to each remote machine process are needed, together with a network *sync* channel to queue the conversations and a network *question* channel.

So, ***JCSP.net*** provides *connections*, which give a high-level abstraction for *two-way* channels for use in networked *client-server* applications.

For example, at the *server* end:

```
Net2OneConnection bar0 = new Net2OneConnection ("ukc.bar0");
```

sets up the infrastructure needed to service a *bidirectional* network connection, registering the name "`ukc.bar0`" (and all relevant type and location information) with the CNS.

At each *client* end of the connection, the statement:

```
One2NetConnection bar0 = new One2NetConnection ("ukc.bar0");
```

sets up the network client infrastructure, getting the server details from the CNS.

Application processes only see *client* and *server* interfaces (rather than *writer* and *reader* ones) to these connections. Each provides *two-way* communications with support for an extended and uninterrupted conversation:

```
interface ConnectionClient {
  public void writeRequest (Object o);
  public Object readReply ();
  public boolean stillOpen ();                 // if unsure
}

interface ConnectionServer {
  public Object readRequest ();
  public void writeReply (Object o);           // keep open
  public void writeReplyAndClose (Object o);   // close
}
```

JCSP.net *connections* are bi-directional and support *any-one* client-server transactions across the network. This means that any number of remote machines may open a named *client* network connection and use it safely – transactions being dealt with atomically at the *server* end [1].

A transaction consists of a sequence of *request-reply* pairs, ending with the server making a *write-request-and-close* reply. If it doesn't know, the client can find out when the transaction is finished by invoking `stillOpen()` on its end of the connection. The server infrastructure locks the server application process to the remote client for the duration of the transaction. Other clients have to wait. Any application process attempt to violate the alternating *request-reply* sequence will raise a (run-time) exception.

A client must commit to read `reply` messages from its server (and, if the transaction was kept open, make a *follow-on* `request`). A server must commit to read *follow-on* `request` messages from its client, if it kept the transaction open with its last `reply`. This is sufficient to keep them synchronised *with no additional system acknowledgements generated across the network* – connection transactions are self-synchronising. Note that connections may not be buffered.

Note, also, that a connection is not *open* until the first reply has been received. Clients may open several connections at a time – but only if all clients opening intersecting sets of connections open them in an agreed sequence. Otherwise, the classic deadlock of *partially acquired resources* will strike.

Many transactions consist only of a *single* request-reply pair. Often this is used to set up an *anonymous* (see Section 2.5) channel or connection for subsequent *private* conversation – see [1] – or simply to get data from a shared server (see Section 2.6). The lack of (system generated) network acknowledgements makes this more efficient than a pair of channels.

Network connections are *ALTable* at their server ends – i.e. '`2One`' connections may be used as *guards* in an `Alternative`. For example, the `Server` process in Figure 3 may wait on events from any of its three networked server *connections* (indicated by the double-arrowed lines), its networked input *channel* and its local input channel.

Finally, we note that core JCSP is extended to support *connections* within an application node (i.e. they do not have to be networked). As for channels, processes only see the *interfaces* to their connections and do not need to know whether they are networked.

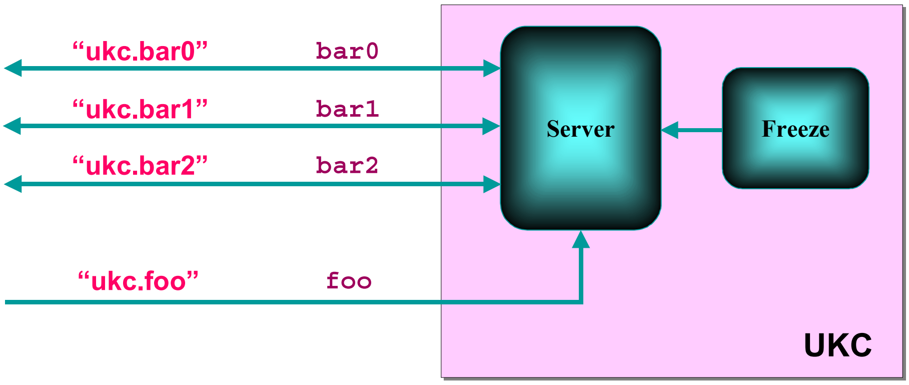

Figure 3. Servicing many events (network connections, network and local channels)

2.5 Anonymous Network Channels/Connections

Networked channels and connections do not have to be registered with the CNS. Instead, *input-ends* of channels (or *server-ends* of connections) may be constructed *anonymously*. For example:

```
Net2OneChannel glik = new Net2OneChannel ();
Net2OneConnection huey = new Net2OneConnection ();
```

Remote processes cannot find them *by name* using the CNS. But they can be told their *location* by communication over channels (or connections) previously set up with the processes that created them. '`Net2`' channels and connections contain location information (*network address* and *VCN*). This can be extracted and distributed:

```
NetChannelLocation glikLocation = glik.getLocation ();
NetConnectionLocation hueyLocation = huey.getLocation ();

toMyFriend.write (glikLocation);
toMyOtherFriend.write (hueyLocation);
```

Remember that your friends may distribute these further!

Processes receiving this location information can construct the *output* (or *client*) ends of the networked *channels* (or *connections*). For example:

```
NetChannelLocation glikLocation =
  (NetChannelLocation) fromMyFriend.read ();
One2NetChannel glik =
  new One2NetChannel (glikLocation);
```

and on, perhaps, another machine:

```
NetConnectionLocation hueyLocation =
  (NetConnectionLocation) fromMyOtherFriend.read ();
One2NetConnection huey =
  new One2NetConnection (hueyLocation);
```

2.6 Process Farms

Figure 4 shows an example use of anonymous channels to build a typical *process farm*:

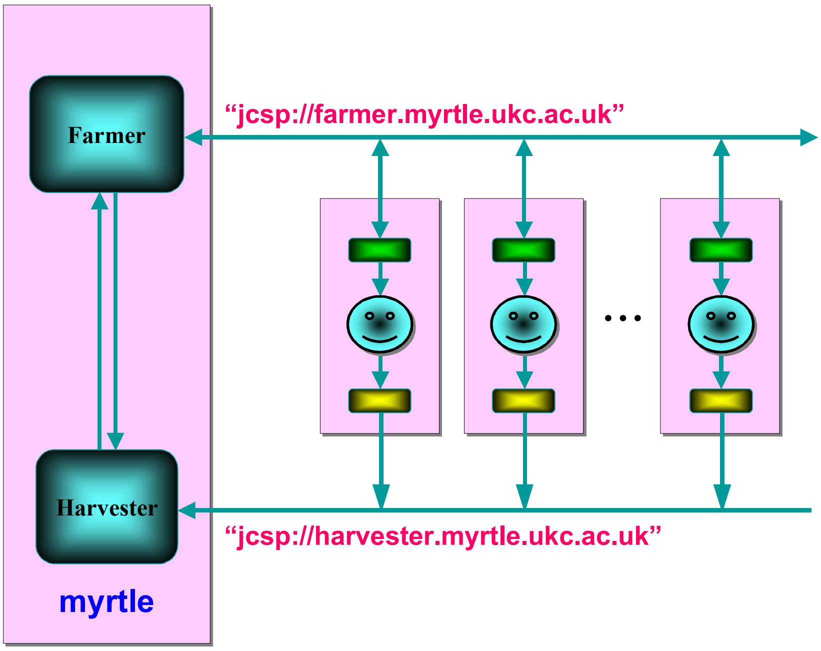
Farmer
"jcsp://farmer.myrtle.ukc.ac.uk"
Harvester
"jcsp://harvester.myrtle.ukc.ac.uk"
myrtle

Figure 4. Process farming

On the (UKC) **`myrtle`** machine, the `Farmer` process services requests from a CNS-published *connection*. The task of the `Farmer` is to generate *independent* packets of work and distribute them, on request, to volunteer `Worker` processes (indicated by the *smiley* faces) residing on different machines. In general, requests need only be `null` packets.

The task of the `Harvester` process is to accept result packets from a CNS-published *channel* (shared and written to by the `Workers`) and assemble them into a solution to the problem addressed by the `Farmer`. The `Farmer` and `Harvester` may need to communicate with each other (and with other processes – not shown in Figure 4). They may also reside on different machines, which would require no change to their implementation.

Neither the `Farmer` nor `Harvester` processes know how many `Worker` processes are available – which number may vary during the course of execution. The `Worker` processes are unaware of each other.

A crucial optimisation is to install a small buffering capacity (say of size 1) into both ends of all network channels. These are indicated by the small boxes on either side of the `Worker` processes in Figure 4 and are standard components provided by JCSP.

So long as it generally takes longer to *solve* a work packet than *receive* it, the next work packet will always be immediately available when required – in the local *farmer connection buffer* (which has pro-actively requested it). So long as it generally takes longer to *solve* a work packet than *transmit* its result, the local *harvester channel buffer* will always be empty when the next result is ready. Thus, the *worker* communications will always return without blocking and almost all processor time will be spent doing the `solve`.

If the above timing conditions are not met, then the problem is not suitable for farming in any case. If they are, this system will yield high levels of efficiency.

2.7 Process Chains (and Rings)

Many problems do not decompose into the *independent* sub-problems necessary for process farming. Typical are those for which the problem space can be divided into adjacent *regions* and processes allocated to work on each region – but for which work, regular exchange of boundary information is needed between neighbouring processes.

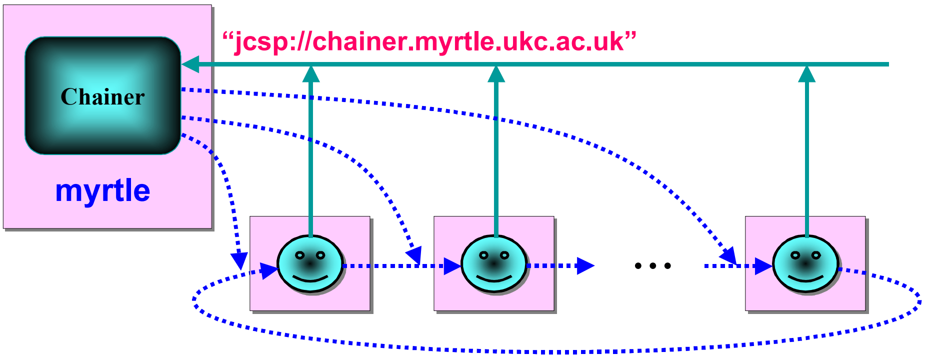

Figure 5. Process chaining

Figure 5 shows a dynamically constructed *chain* (or, in this case, *ring*) of networked processes. The `Chainer` process is only used to set up the chain and plays no part in the resulting computation (unless nodes are lost or added). It services a CNS-published *channel*, whose name is known to all volunteers for the chain gang. All other channels are *anonymous* – shown in Figure 5 as dotted arrows.

Each `Worker` node creates an anonymous network *input* channel (with a small buffering capacity – say 1) and sends its *location* to the `Chainer`. The latter, on receipt of such a location, saves it and creates the *output-end* of the channel. When the `Chainer` has enough such volunteers, it closes registration and – using the channels just received – sends to each `Worker` the location of the output channel to use. Simple *on-the-fly* construction code is:

```
final int N_NODES = ... ;                  // (e.g. from command line)

Net2OneChannel fromWorkers =
  new Net2OneChannel ("jcsp://chainer.myrtle.ukc.ac.uk");

NetChannelLocation lastL = (NetChannelLocation) fromWorkers.read ();
One2NetChannel lastC = new One2NetChannel (lastL);

for (int nWorkers = 1; nWorkers < N_NODES; nWorkers++) {
  NetChannelLocation nextL = (NetChannelLocation) fromWorkers.read ();
  One2NetChannel nextC = new One2NetChannel (nextL);
  nextC.write (lastL);
  lastL = nextL;
}

lastC.write (lastL);                       // completes the network ring
lastC.write (new int[] {0, N_NODES});      // final ring synchronisation
```

The `Chainer`'s work is now done – unless it is to coordinate the removal or addition of `Worker` nodes. More clever code may connect the nodes together in a way that takes into account *neighbourliness* (e.g. inter-node latency and bandwidth). However, that is not needed for a cluster of nodes on a symmetrically switched *ethernet* (Section 6.1).

The *final ring sync* is to prevent premature execution of the `Worker` processes. The *first* communication a `Worker` node receives is the network location of its output channel. It must not start working yet, though, since an application communication may then race the network ring completion message on that `lastC` channel. Instead, it waits for a *second* communication which must be forwarded round the ring once – starting from `lastC`.

Here is example `Worker` code:

```
One2NetChannel toChainer =
  new One2NetChannel ("jcsp://chainer.myrtle.ukc.ac.uk");

Net2OneChannel in = new Net2OneChannel ();
NetChannelLocation inLocation = in.getLocation ();

toChainer.write (inLocation);                // only used once

NetChannelLocation outLocation = (NetChannelLocation) in.read ();
One2NetChannel out = new One2NetChannel (outLocation);

int[] info = (int[]) in.read ();             // wait for ring sync

final int MY_ID = info[0];                   // don't really need this
final int N_NODES = info[1];                 // don't really need this

info[0]++;
if (info[0] < info[1]) out.write (info);     // pass on ring sync

new WorkProcess (MY_ID, N_NODES, in, out);   // run the application
```

The first two parameters to the application `WorkProcess` are luxuries – unless we want to cater for the dynamic *retirement* and/or *introduction* of `Worker` nodes (or, of course, the application being run really needs it). Finally, note that the application `WorkProcess` itself takes no part in setting up the ring network and remains unchanged from its *uni-processor* version.

The *final ring sync* would not be needed if each `Worker` node made *two* network (input) channels – one for its ring input and one from which it received the location of its ring output channel (and, possibly, subsequent control interventions) – and sent the locations of *both* of them to the `Chainer` (in a single communication, wrapped in an array). In this case, the `Chainer` would make a network output channel of *only one* of them and use that to send the appropriate ring output location (and, possibly, subsequent control signals).

Of course, the ring network could be set up *directly* using CNS-registered *named* channels – in which case no `Chainer` is needed. However, in that case each worker node *will* need to know the size of the ring and its place in it (to generate, register and use unique channel names). The `Worker` code, though, becomes quite trivial:

```
final int MY_ID = ... ;                      // obtain somehow
final int N_NODES = ... ;                    // (e.g. command line)

final int NEXT_ID = ((MY_ID + 1) % N_NODES);

Net2OneChannel in = new Net2OneChannel ("node-" + MY_ID);
One2NetChannel out = new One2NetChannel ("node-" + NEXT_ID);

new WorkProcess (MY_ID, N_WORKERS, in, out);  // run the application
```

However, the freedom afforded to the `Worker` nodes – from not having to know (in advance) the structure of the distributed application in which they are going to participate – makes the use of *anonymous* channels very attractive. Also, we only have a single point of dependency on the CNS (for the "`jcsp://chainer.myrtle.ukc.ac.uk`" registration) and it's good to be independent!

Two rings, giving *bi-directional* communication between nodes, can be set up with only a little extra work. For the applications envisaged at the start of this subsection, that would be better than a single ring of `Connections` – which also could provide *bi-directional* communication. This is because the latter would *serialise* the exchange of data between the nodes – i.e. first data flows one way (the *request*) and, then, the other (the *reply*). With a pair of *opposing direction* rings of `Channels`, the exchange can happen in *parallel*.

3 The DANISH Benchmark Suite

The applications presented here are part of the benchmark suite *DANISH* (*D*anish *A*pplications for *N*etworked *I*PC and *SH*ared memory). It is important to stress that *DANISH*[14, 15] is not designed to benchmark Java Virtual Machines, nor is it designed to test the performance of physical machines. The idea behind *DANISH* is to provide relatively simple means to test parallel programming APIs for Java, whether they are targeted for distributed or shared-memory architectures. The primary measure for *DANISH* is *scalability* rather than actual execution time or some measure of computations per timeframe. *Speedup* is therefore chosen to enable the best possible comparisons between published results and across different platforms.

Our own previous work with designing message-passing and distributed shared memory systems identified the development of benchmark applications as a major contributor to frustration. Once a parallel programming toolset and API has been developed, it is unsatisfactory not to be able to test its performance straight away.

Finding and porting suitable applications is often much work and if one wishes to compare the performance of a new system to existing ones, the applications must also be ported to all toolsets that are used for comparison. Such a port by the authors of a new toolset, is naturally distrusted by readers of the published results (*'Are the comparison APIs utilised properly?'*, *'Are the applications chosen to favour the new API?'*, etc.).

Existing benchmarks are often not viable; they may be too specific towards testing machine performance, or they may not be diverse enough in the type of applications that they include. *DANISH* seeks to provide a suite of different applications, grouped into application types, and kept simple enough so that porting is simple. In addition, benchmark implementations are provided for a set of existing APIs, which can be used as the reference base – thus reducing the problem of credibility of the results obtained for the new toolsets.

Users must port the *DANISH* benchmark suite to use a new API by hand. This can be done by parallelising the sequential version in some appropriate (perhaps novel) way or by using one of the existing parallel versions as a template. This approach was chosen over a version where the user simply provides his or her own implementation of a set of generic primitives, so as not to inhibit the range of APIs that may be tested and discourage researchers from utilising radically different technologies. The applications in *DANISH* are all well documented, both as an abstract description and inside the provided code. Wherever applicable, there is the option of a graphic output that may help debugging if a port goes wrong. We seek to keep all applications in *DANISH* as small as possible so as further to ease porting. The parts of the code that require communication are kept isolated from parts that do not – to the largest degree possible. All source codes are available [15].

4 Existing Parallel Toolsets

To compare the use and performance of JCSP with current *state-of-the-art* tools for parallel computing, we have included versions of the chosen benchmarks programmed using *mpiJava* and *TSpaces*.

4.1 mpiJava

The *mpiJava*[16] version that is presented here is an interface to a native code MPI version, the MPICH[17] implementation of MPI. MPI, and with it *mpiJava* supports a multi-process execution model, in fact the number of processes that should be used are specified at load-time. The individual processes may then make calls to the runtime environment to inquire on the number of processes in the execution as well as their own order in the process range.

4.2 TSpaces

TSpaces[18] is structured DSM (*Distributed Shared Memory*) system similar to JavaSpaces[19] – both of which draw heavily on Linda[20]. Rather than supporting communication between explicit processes *TSpaces* supports writing to and reading from an associatively addressed shared memory area. Since this is a memory model and not a communication model, there is no *process* concept provided with *TSpaces* and all processes must be started by other means. An interesting feature of this model it that, being a memory model, processes may be *temporally disjoint* (e.g. one process may read data written by a process that has already terminated – even before the reading process had begun). However, this effect can also be obtained in JCSP (by using *buffered* channels) and in MPI (by *asynchronous* communication).

TSpaces is a very simple API to use, and except for the lack of a remote process creation option, porting code to *TSpaces* is very straightforward.

5 Applications

The applications we have chosen to test the performance of *JCSP.net* are all taken from the *DANISH* benchmark-suite. We have chosen a set of applications that represent distinct behaviour found in high performance applications, with a special focus on data-dependency and communication patterns.

5.1 n-Body Problem

The chosen *n-body* algorithm is a trivial $O(n^2)$ complexity simulation of celestial bodies.

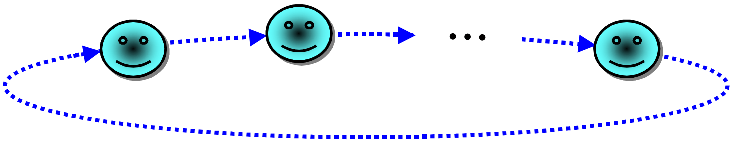

Figure 6. N-Body simulation in a ring

The JCSP implementation divides the bodies amongst worker processes. Each worker has channel connections to two others so that they form a ring (as shown in Figure 6). The JCSP mechanisms for establishing this topology are described in Section 2.7 (*process chaining*) and Figure 5.

In parallel, each worker sends (and receives) around the ring a copy of the relevant information (e.g. positions and masses) pertaining to the bodies it was *originally given*. In parallel with all this, each worker also computes the forces *on its given bodies* by each set of *passing bodies*. After `(p-1)` such parallel operations, where `p` is the number of workers in the ring, all bodies in all workers have had the forces on them calculated with respect to all other bodies and their velocities and positions are updated (with each worker able to compute independently in parallel). After this another iteration may begin.

The MPI version is based on all workers broadcasting their set and updating their bodies with all the incoming bodies. The *TSpaces* version is implemented by each worker writing its bodies to the shared *tuple-space* and all reading from the shared space.

5.2 Successive Over-Relaxation

Successive Over-Relaxation, SOR, is a frequently used technique for solving very large systems of partial differential equations by successive approximations. The general idea is to approximate each element in a matrix by reference to its nearest neighbours until the sum of all changes within one iteration converges below a given *epsilon* value.

The *red-black checker* pointing version of SOR, shown in Figure 7, returns identical results for the same system of equations; independent of the actual computing environment, while at the same time providing sufficient parallelism that real speedup can be achieved. The equation system is divided into alternating red and black points in a *checker-board* fashion. Updating a red point depends only on black neighbouring points and *vice versa*. Using this, an algorithm is derived where each worker-process updates all its red points and then exchanges red border point values with its neighbours. Each worker then updates its black points and repeats the communication for the black points. At the end of each iteration, the global change in the system is calculated. This continues until the change is below the given *epsilon*.

The SOR application is parallelised by striping the matrix, using one stripe per worker. Each worker then has to exchange its left and right columns with its neighbours in each iteration – except of course for the first and last workers, which have no left or right neighbours respectively.

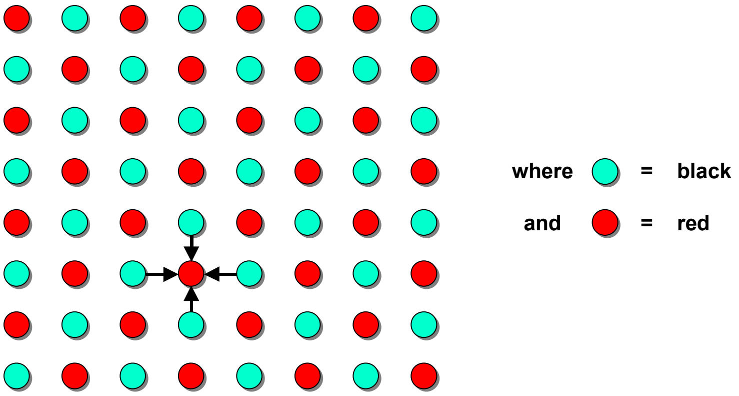

Figure 7. Red-Black checker-pointing in SOR.

The JCSP version is implemented using a two-way pipeline as shown in Figure 8. This also is set up as described in Section 2.7.

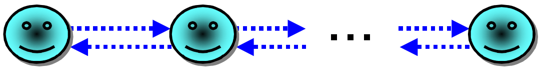

Figure 8. Double pipeline architecture of the SOR application for data-exchange.

The calculation of the *sum-of-all-changes* that is used as the halting criteria is implemented using a JCSP library function that builds a tree of the workers and assembles and spreads the sum along this tree as shown in Figure 9.

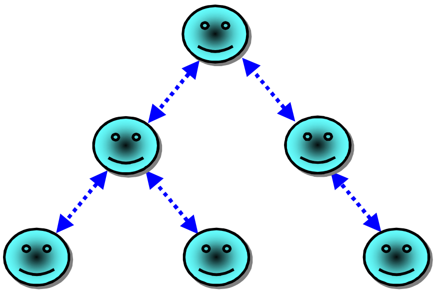

Figure 9. Summation tree for finding the total change in all workers

The *TSpaces* version is implemented by each worker storing its top and bottom row in *tuplespace,* once per iteration. The global sum is calculated using a *setup* with two *tuples*; (“`Sum`”, `int cnt, float val`) and (“`Result`”, `int round, float val`). Each worker takes the “`Sum`” tuple from the tuplespace, adds a partial sum to the value and increases the counter. If the worker is the last to write its sum, it writes back a new zero version of the “`Sum`” tuple and writes out the new “`Result`” tuple. Otherwise, it waits to read the “`Result`” tuple.

The MPI version of this application is straightforward as it is the kind of application for which MPI is well suited. The workers exchange their rows using asynchronous message-passing and find the global change using the `MPI_Globalreduction` function.

5.3 Travelling Salesman Problem

The Travelling Salesman Problem, TSP, is a classic representative for the class of global optimisation problems. The TSP solution we use in this work is a *depth-first branch-and-bound* algorithm. This makes the parallel version different from the other applications we use by the fact that a *static division* of the work would result in a highly unbalanced execution. Thus the parallel TSP is implemented as a *bag-of-tasks* (i.e. ‘farming’) application. This paradigm that does not come naturally to the SPMD programming paradigm around which MPI is designed, while it suits *Tuple-Space* models such as JavaSpaces and *TSpaces* very well and is easily modelled with JCSP.

The parallel TSP is implemented as a global *master* process and set of *worker* processes on each processor. Each worker communicates with the master to retrieve jobs and submit results. A job is represented as a set of cities that have already been placed and a set that needs to be placed, i.e. a sub-tree. Once a new candidate to the shortest route is found, the master-process broadcasts this new *bound variable* to all of the workers.

The TSP, therefore, is not a straightforward farming application. The individual jobs performed by the workers are not completely independent from each other – results from each work packet may have determined better shortest path candidates and these must be distributed to running workers, where they may impact the computation.

We use a combination of a simple version of a process *farm* (with the `Master` process combining the roles of both farmer and harvester) and a process *chain* (but without the chain channels!) – see Figure 10. The mechanism for setting this up are as described in Section 2.7, minus the establishment of the actual chain.

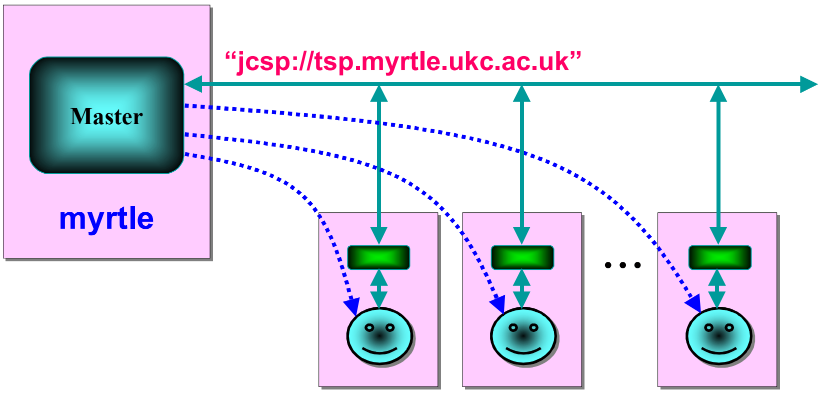

Figure 10. A controlled process farm.

The worker processes request jobs by sending their previous results (initially `null`). Each recursive level in the iterative tree-spanning search algorithm is started with a poll on their (anonymous network) input channel to check if a new bound variable has been received.

The *TSpaces* and MPI versions use a similar approach, however without a polling mechanism. Instead, they use an approach where the bound is only updated with each new job the worker receives.

mpiJava does include an *asynchronous* read that could be used to receive updates at runtime. When we used this technique, problems occurred with badly formatted packages that caused the application to crash. There were similar problems with returning the result from a job because, once more objects were sent to the master-process, it could not distinguish between them. The application would then crash from a serialisation error. This was fixed by returning only the length of the found path and not the complete path. In principle, this means that the *mpiJava* version returns much less data than the *JCSP.net* and *TSpaces* versions but, in reality, this is probably of minor consequences.

TSpaces does include a '`tryToRead`' operation that may be used for polling for the updated bound-variable. However, using this takes just as long as reading a shared version of the bound-variable each time, and doing so brought the application to almost a complete stop.

6 Performance

6.1 Experimental Platform

The test machine is a cluster of 16 dual Pentium III, 450 MHz nodes, each with 128-MB memory. The machines are connected via Fast-Ethernet through a switch with a back-plane capacity of 2.1 Gb/sec, e.g. enough to service all NICs at the same time. The JVM used is Sun's Java for Linux version 1.3.1 and the OS is Linux 2.2.14.

6.2 Latency Hiding

JCSP has the strong advantage that extra concurrency (i.e. channels and processes) may easily be added within a processing node and that that concurrency conforms to exactly the same model used for the physical parallelism of the distributed system. Further, since processes interact only with channels and not with other processes, no re-design or re-programming of existing processes is needed.

For example, we may add trivial buffer processes between the network channels and worker processes in each physical node. Such buffers are shown in Figures 4 and 10 (for *process farms*). The `Worker` nodes in Figure 5 (*process chains/rings*) are not currently buffered. To do so, just change one line in the `Worker` code given in Section 2.7:

```
new WorkProcess (MY_ID, N_WORKERS, in, out);   // run the application
```

into:

```
One2OneChannel p = new One2OneChannel ();
One2OneChannel q = new One2OneChannel ();

new Parallel (                                 // run the application
  new CSProcess[] {
    new BufferProcess (1, in, p),              // high priority
    new WorkProcess (MY_ID, N_WORKERS, p, q),  // mid priority
    new BufferProcess (1, q, out)              // high priority
  }
).run ();
```

and notice that the `WorkProcess` itself needs no modification.

We must beware that adding buffers can alter the *semantics* of a system (i.e. cause different results to be produced). However, this is not the case for the simple interaction patterns followed by all the applications presented here.

The point of all this buffering is as follows: so long as the underlying JVM and operating system does sensible things regarding thread priorities … and so long as we take care to ensure those buffers are not starved … and so long as the hardware (NICs, processors etc.) enables it, *we can drive the network links in parallel with productive computation*. Thus, the latency cost of network communication can be considerably hidden – and sometimes completely eliminated. This issue was also discussed Section 2.6 (in the context of process *farms*).

Buffering is such a common and useful technique that JCSP provides (several varieties) of buffer plug-ins for its core (i.e. *non-networked*) channels. ***JCSP.net*** *will* provide buffer *plugins* for networked channels – currently, these are only available for networked *input* channels. In which case, the above `Parallel` construct can be removed and the original *one liner* restored – the buffers being set up in the construction of the `in` and `out` channels. The end result is logically the same, but with reduced code complexity and thread context-switching.

The MPI version uses the *asynchronous* send operations (`Isend`) to obtain the buffered communications that the JCSP solutions use. *TSpaces* do not have such functionality since this is a shared memory model. Instead, the *TSpaces* version uses a *write-early* and *read-late* approach that tries to ensure that, once a process is ready to read a tuple from another process, this tuple is likely to exist in the *Tuplespace*.

6.3 n-Body problem

The n-Body benchmark was chosen to demonstrate the consequences of the lack of a broadcast mechanism. The results, however, are quite surprising (Figure 11) since the ring-approach we use with JCSP significantly outperforms the broadcast based MPI version and the shared data version in *TSpaces*. At this time of writing, we have not had the time to re-implement the MPI and *TSpaces* versions with an equivalent ring algorithm, which we intend to do to better understand this quite surprising result.

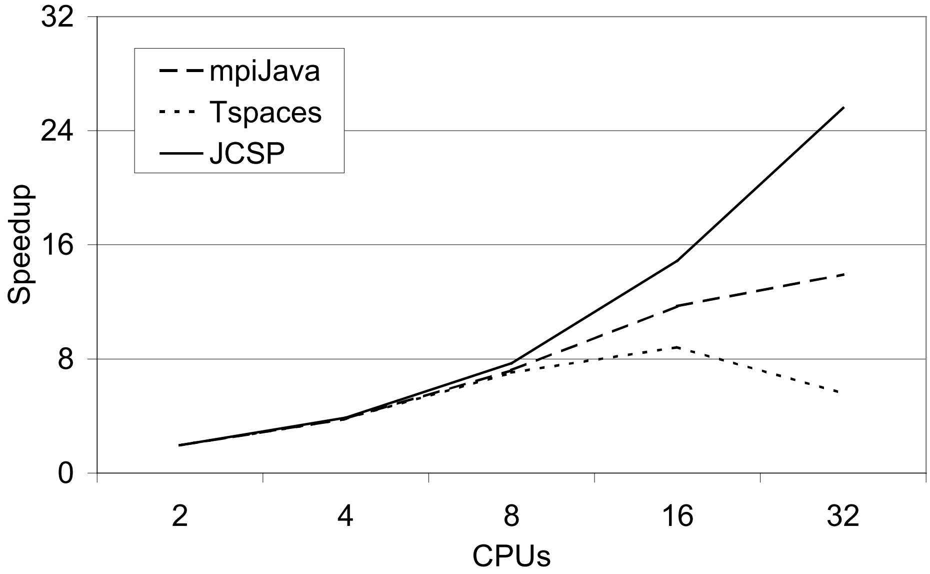

Figure 11. The performance of the n-Body benchmark (n = 10000)

For 10000 bodies (*serial time* = 1044 seconds), *JCSP.net* achieves a speedup of 26 with 32 CPUs. This is a quite respectable result, especially considering that there is a large amount of object serialisation taking place. In fact, the *number* of serialisations that take place per worker for each iteration grows linearly with the number of workers (and equals the number of communications). The *volume* of data being serialised and communicated per worker, however, stays the same (for a large number of workers).

6.4 Successive Over-Relaxation

The SOR application (7000 x 7000 matrix, *serial time* = 253 seconds) loaded the *TSpaces* server so heavily that we did not succeed in getting the application to run on 32 CPUs. Even without this, it is obvious that *TSpaces* is not well suited for this kind of application – a likely cause being contention for its centrally serviced *Tuplespace*.

SOR is a typical matrix application and is, therefore, the kind of application for which MPI is well-suited. With this in mind, the performance of JCSP is quite respectable (Figure 12), especially considering that MPI provides a *built-in* function for doing the necessary global sum. This function is implemented in the *NMI* MPI-layer, while JCSP performs it using standard channel communication.

The *mpiJava* curve tops at a speedup of 28 using 32 CPUs – with *JCSP.net* achieving a speedup of 25 using the same.

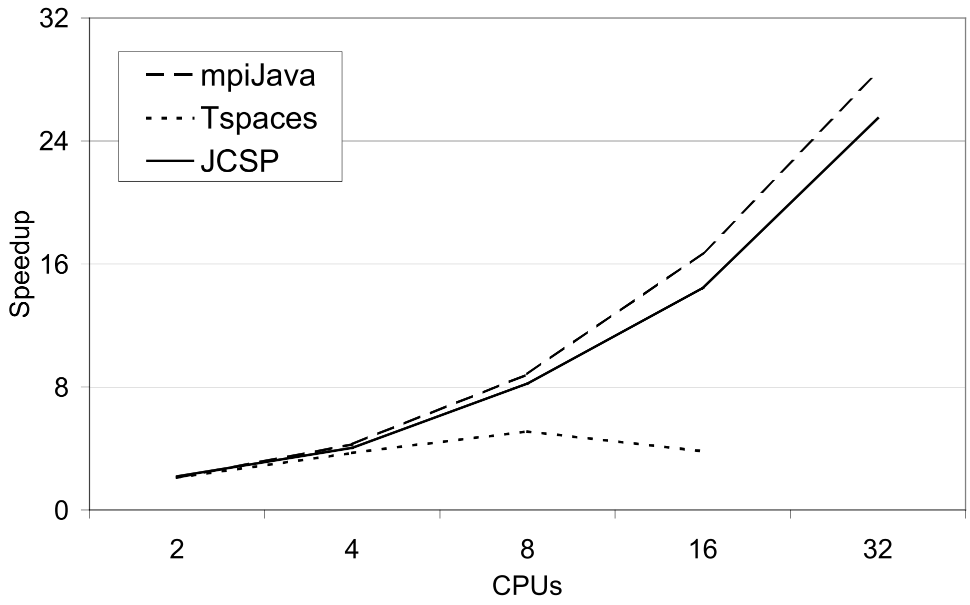

Figure 12. The performance of the SOR benchmark (7000 x 7000 matrix)

6.5 Travelling Salesman Problem

The *branch-and-bound* approach adopted in the TSP solution is interesting from a parallelisation perspective, since it may mean that the parallel solution may in fact end up performing more or less work than the sequential version. The approach that is used with MPI and *TSpaces*, however, cannot result in less work than the sequential version. One should accordingly expect the JCSP version to perform better than the two others. This is certainly true relative to *TSpaces*, but the MPI version performs significantly better than the JCSP one. We are still in the process of identifying bottlenecks in the JCSP version.

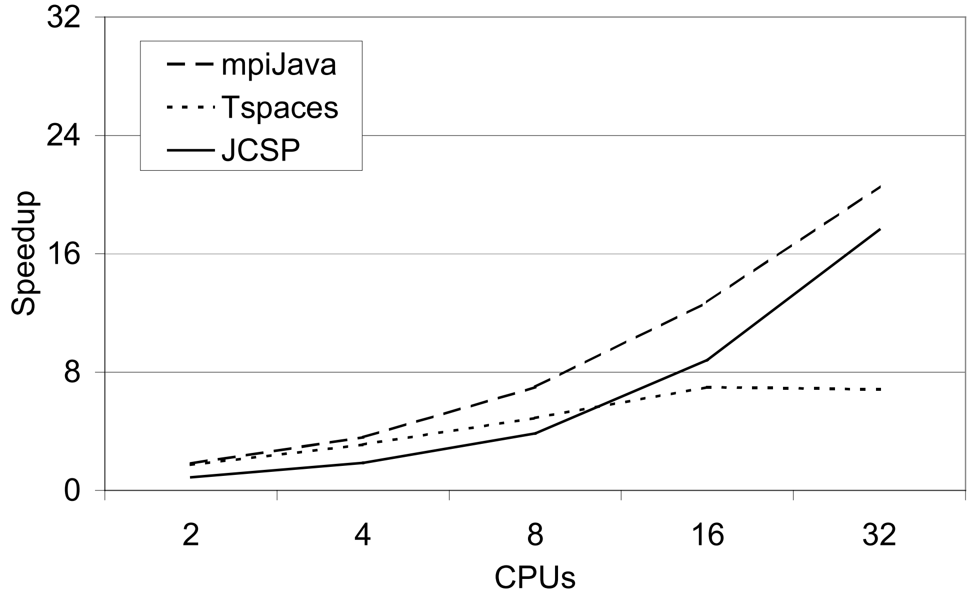

Figure 13. The performance of the Travelling Salesman Problem (15 cities)

The MPI version seemed to identify an error in the *mpiJava* package. When many processes were writing to the server, the serialisation of the objects went wrong which in turn resulted in an *"Object not serializable"* error. Based other experiences, we suspect that this problem is inherent in our JVM and not located in *mpiJava*. So, we had to modify the Java version to return only the length of the route and not the complete route.

The TSP benchmarked searches routes for 15 cities (*serial time* = 374 seconds). The value and order of the cities in the list does impact on performance, so these were kept the same for all trials.

Using 32 CPUs, *mpiJava* achieves a speedup of 21, *JCSP.net* reaches 14 and *TSpaces* never gets above 7 (see Figure 13). JCSP follows the same speedup curve as MPI – but does so from a much poorer origin. At two CPUs, we have an actual negative speedup of 0.9 (i.e. a CPU utilisation of 0.45). This is obviously not a good result and we will be taking another look at this.

7 Conclusions and Future Work

This paper presents some of the key facilities of ***JCSP.net*** and the JCSP mechanisms used to implement them. Application concepts include *any-one* networked channels and *connections* (that enable extended *client-server* conversations), a *Channel Name Server* (CNS) for the dynamic construction of application networks, and *anonymous* channels and connections (that evade the normal CNS registration). For a fuller exposition – including mobile *processes* (a.k.a. *agents*), mobile *channel read-ends*, mobile *server connection-ends* and an outline of the implementing JCSP infrastructure – see [1].

In this work, we show *preliminary* results that indicate that high performance may be achieved *simply* with this model. In fact, ***JCSP.net*** consistently outperforms the shared memory approach found in *TSpaces* and compares favourably with *mpiJava* (even though that is based on *NMI*, where much of the communication takes place outside the JVM). Once one comes to terms with the simplicity of CSP-based concurrency, it is quite attractive to use for high performance applications.

JCSP.net[1] is a very recent extension to JCSP[5, 6] and no serious effort, other than that put into its fundamental design for lightness, has been made on optimisation. ***JCSP.net*** is now being developed and supported commercially (by Quickstone Ltd.[21], under license to the University of Kent) in collaboration with its originators[22]. Results reported here are only for the *alpha* release of *JCSP.net* and should improve as implementations mature.

Some questions remain to be answered – especially the anomaly in the two CPU performance of our TSP/JCSP implementation. The optimisation described in Section 2.3 (that promotes the overlapping of computation and communication) is not in the current code. Other approaches to this problem will be considered including distributing new lower bounds by chaining the *workers* into a ring (rather than via the central *master* process).

A major force in ***JCSP.net***, which it derives from CSP, is the localisation of process semantics behind a channel interface. Processes do not care whether their connected channels are local or networked. Hence, decisions can be left late (and changed) as to which processes run where – they can even be taken at runtime.

To investigate these issues further, we shall be trying out a set of applications using *Clusters of Multi-Processors* with 2, 4 and 8 CPUs per node. ***JCSP.net*** also allows communication between any machines connectable via TCP/IP. Thus, widely distributed (*Grid-like*) systems using ***JCSP.net*** will be investigated, where the application topology puts itself together dynamically.

JCSP.net is part of a larger project on language design, tools and infrastructure for scalable, secure and simple concurrency. In particular, we are developing the multi-processing **occam** language with various kinds of dynamic capability (see the ***KroC*** website [11] *and elsewhere in these proceedings*) that match the flexibility available to Java systems, but which retain strong semantic checks against concurrency errors (such as *race hazards*) and ultra-low overheads for concurrency management (some two to three orders of magnitude lighter than those accessible to Java). The Java and **occam** concurrency work feed off each other in many ways – there will, for example, be a ***KroC.net*** [23].

References

[1] P.H.Welch, J.R.Aldous and J.Foster. *CSP Networking for Java (JCSP.net)*. In 'Global and Collaborative Computing' Workshop Proceedings, ICCS 2002, Lecture Notes in Computer Science, Volume 2330, pp. 695-708. Springer-Verlag. April, 2002.

[2] C.A.R.Hoare. *Communicating Sequential Processes.* CACM, 21-8, pp. 666-677, August 1978.

[3] C.A.R.Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[4] A.W.Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, ISBN 0-13-674409-5, 1997.

[5] P.H.Welch and P.D.Austin. *The JCSP Home Page*. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/. 2002.

[6] P.H.Welch. *Process Oriented Design for Java – Concurrency for All*. In: 'PDPTA 2000', Volume 1, pp. 51-57. CSREA Press, ISBN 1 982512 22 X, June 2000.

[7] D.Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns*. The Java Series, Addison-Wesley, section 4.5, 1999.

[8] P.H.Welch, *Java Threads in the Light of occam/CSP*. In 'Architectures, Languages and Patterns for Parallel and Distributed Applications', WoTUG-21, pp. 259-284, IOS Press (Amsterdam), ISBN 90 5199 391 9, April 1998.

[9] P.H.Welch. *Parallel and Distributed Computing in Education*. In J.Palma et al. 'VECPAR'98', Lecture Notes in Computer Science, vol. 1573, Springer-Verlag, June 1998.

[10] Inmos Limited. *occam2.1 Reference Manual,* Technical Report. http://wotug.ukc.ac.uk/parallel/occam/parallel/occam/documentation/. 1989.

[11] P.H.Welch, J.Moores, F.R.M.Barnes, D.C.Wood. *KRoC Home Page*. http://www.cs.ac.ukc.ac/projects/ofa/kroc/. 2002.

[12] Inmos Limited. *occam3 Reference Manual,* Technical Report http://wotug.ukc.ac.uk/parallel/occam/parallel/occam/documentation/. 1992.

[13] M.D.May, P.W.Thompson, P.H.Welch: *Networks, Routers and Transputers*. IOS Press, ISBN 90 5199 129 0 (1993).

[14] Brian Vinter, *Embarrassingly Parallel Applications on a Java Cluster*, Proceedings of 'The European Conference on High Performance Computers and Networking (HPCN)', Lecture Notes in Computer Science, Volume ????, pp. 129-149. Springer-Verlag. 2000.

[15] Brian Vinter, *DANISH Home Page*. http://DANISH.imada.sdu.dk/. 2002.

[16] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. *mpiJava: An Object-Oriented Java interface to MPI*, Presented at the 'International Workshop on Java for Parallel and Distributed Computing', IPPS/SPDP 1999, San Juan, Puerto Rico, April 1999.

[17] *MPICH Home Page.* http://www-unix.mcs.anl.gov/mpi/mpich/

[18] *TSpaces Home Page*. http://www.almaden.ibm.com/cs/TSpaces/

[19] Eric Freeman, Susanne Hupfer, Ken Arnold. *JavaSpaces(TM) Principles, Patterns and Practice*, SUN MicroSystems.

[20] David Gelernter, *Generative Communication in Linda*. ACM Transactions on Programming Languages and Systems, vol. 7. 1985.

[21] Quickstone Technology Limited. http://www.quickstone.com .

[22] P.H.Welch. *Concurrency Research Group Home Page*, Computing Laboratory, University of Kent at Canterbury. http://www.cs.ukc.ac.uk/research/groups/crg/. 2002.

[23] M. Schweigler. *The Distributed occam Protocol – Channels over the Internet*. MSc Dissertation, Computing Laboratory, University of Kent at Canterbury. September 2001.