Performance of Parallel Communication and Spawning Primitives on a Linux Cluster

D. J. Johnston, M. Fleury, M. Lincoln, and A. C. Downton*

Abstract

The Linux cluster considered in this paper, formed from shuttle box XPC nodes with 2 GHz Athlon processors connected by dual Gb Ethernet switches, is relatively easily constructed, but, while effective as a throughput engine, may result in disappointing results when running explicitly parallel software if weakly-performing communication mechanisms and process spawning are selected. This paper carefully compares the implementations of communication and spawning primitives in MPICH-2, openMosix, and Linux Remote Procedure Call, forking, and various lower-level communication mechanisms. The test selection compares the provision of both a message-passing library, and a single system image software package, with direct use of lower-level primitives. The information in the paper will be of interest to those considering the use of one of the well-known packages, or directly writing their own distributed applications, or constructing a distributed language by layering on top of an existing set of parallel primitives. The results expose a ranking in terms of process spawning and a similar ranking of communication software performance. They reveal poor performance in certain circumstances, well below the hardware specification, which it is as well that the developer is aware of. In general, the paper emphasizes the importance of efficient transport software to cluster machines.

1 Introduction

Developers of new real-time parallel computing libraries (harnesses) on clusters must make a compromise between speed of development and performance. In some cases, harnesses have

*The authors are affiliated to the University of Essex, Department of Electronic Systems Engineering, UK, Wivenhoe Park, Colchester, CO4 3SQ, United Kingdom, tel: +44 - 1206 - 872817, fax: +44 - 1206 - 872900, e-mail: {fleum,acd}@essex.ac.uk

been layered upon pre-existing higher-level spawning and communication packages such as the well-known PVM [1]. For example, the array programming language ZPL [2], though following the shared-memory model, for distributed memory implementations employs either PVM or MPI for communication. In other cases, lower-level primitives, such as Unix `rsh` and the socket application programming interface (API), are utilized, as is the case for PVM itself. The investigation recorded in this paper started with the need to extend a uni-processor concurrent programming language, Concurrent Meta-Language (CML) [3], to a cluster. A way of translating CML from its underlying functional language form to the C programming language code had already been devised. Existing parallel languages did not present the event-based communication model of CML [4] which make CML suitable for cluster and grid computing. Rather than simply select any system that was available, the strategy adopted was to benchmark the available parallel primitives. Therefore, the information recorded in this paper will be of interest to those considering the use of one of the well-known packages, or directly writing their own distributed applications, or constructing a distributed language by layering on top of an existing set of parallel primitives, and not simply to CML implementation.

The three systems selected for investigation were MOSIX, MPI-2 and the underlying Linux mechanisms: Remote Procedure Call (RPC), fork, and a variety of Unix communication mechanisms such as pipes, fifos, messages, and sockets. These systems were selected for trial tests as they are very different types of software. MPI-2 [5] is a well-known message-passing system consisting of libraries, servers and utility programs running on top of an operating system such as Linux, and, hence, acts as a portability layer. Though PVM and MPI are discussed by the originator of Beowulf computers [6] as bedrock systems, PVM's originators suggest [7] that MPI is more suitable for homogeneous systems. MOSIX [8] consists of extensions to the Linux kernel[1] in order to transparently migrate processes to remote nodes, while presenting a single system image. For a survey of cluster load-balancing systems which also support parallelism, refer to [10]. Though MOSIX can be employed as a throughput engine, it has been recommended by its authors [11] for parallel applications on cluster computers, for example molecular dynamics simulations. Linux, although the underlying operating system in both instances, itself has lower-level mechanisms, such as the socket API and RPC[2] [12, 13], that can directly support parallel processing. Other than performance, an application implementor should also use criteria of ease of configuration, programming and debugging convenience, which is why high- and lower-level constructs have been included in the comparison. Notice that it is possible to combine MPI and Linux sockets, but that MOSIX is incompatible with

[1] Prior versions of MOSIX [9] were written for other versions of the Unix operatings system.

[2] In fact, RPC is usually layered on top of the socket API.

the other two, as it imposes its own version of all socket system calls.

There is considerable literature on the benchmarking of parallel computers [14]. However, benchmarking is normally concerned with comparing the performance of computers rather than the software which runs on them. The results reported in [15], which compare CORBA, the ACE C++ communication wrapper library, and C using sockets, across an 155 Mb/s ATM network[3], is closer to the intentions of this paper, which is better described as benchmarking software. In [15], for a medical storage area network application, CORBA control signals are integrated with socket API calls for bulk data transfer. The results in [15] identify a "disparity between network channel speed and end-to-end application throughput", with only 40% of the ATM network's bandwidth being utilized. The results herein are another example of what has been called the "throughput preservation problem" [16] with reference to lower-level mechanisms such as Unix System V STREAMS pipes and messages, or BSD Unix sockets.

The paper is organized as follows. The next section provides a brief description of the cluster machine on which the tests took place and gives more information on the installed software. Section 3 considers the experimental framework, while Section 4 reports and analyzes the tests. Finally, Section 5 draws some conclusions.

2 Linux cluster

The cluster employed consists of thirty-seven processing nodes connected with two Ethernet switches [17] amusedly designated the "Heap"[18]. Each node is a small form factor Shuttle box (Model XPC SN41G2) with an AMD Athlon XP 2800+ Barton core (CPU frequency 2.1 GHz), with 512 KB level 2 cache[4] and dual channel 1 GB DDR333 RAM In Figure 1, the nodes are connected via two 24 port Gigabit (Gb) Ethernet switches manufactured by D-Link (model DGS-1024T). Each switch is non-blocking and allows full-duplex Gb bandwidth between any pair of ports simultaneously. Each switch can be connected via independent network cards in the server, to allow the cluster to be partitioned into two, designated "big" and "little heap" in Figure 1. Alternatively, the two switches can be linked together via a Gb port, though obviously communication between nodes on different switches becomes blocking. The switches are un-managed and, therefore, unable to carry "jumbo" 9000 B Ethernet frames (which would lead to an increase in communication efficiency of about 1.5% and, more significantly, a considerable decrease in frame processing overhead[19]).

To make maintenance and cluster-wide propagation of configuration changes easier, at

[3]The abbreviation 'b' for bits is used throughout this paper.

[4]The abbreviation 'B' for bytes is used throughout this paper.

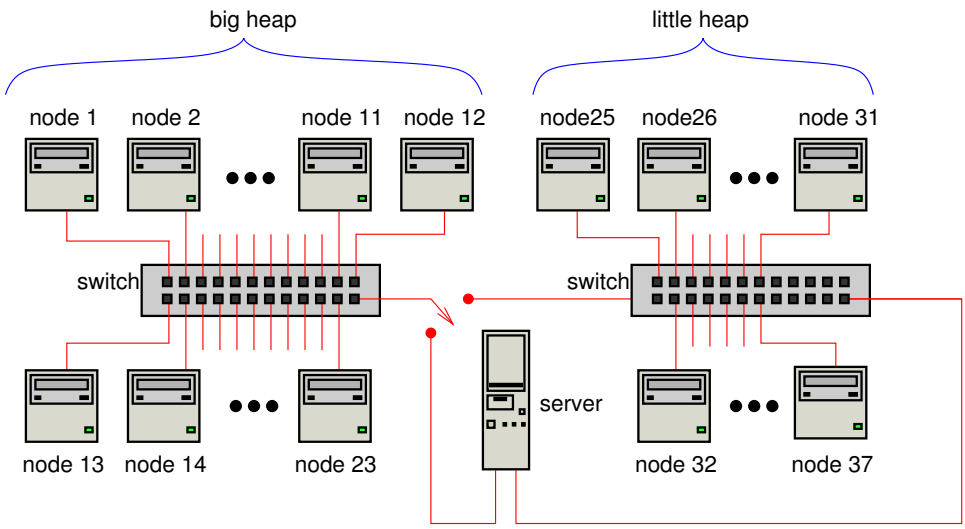

Figure 1: Schematic diagram of the Linux cluster, showing system partitioning and fileserver

boot time all nodes transfer their root file system from a file server to the local disk, while other file systems are accessed via the Network Filing System (NFS) [20].

The bandwidth cross-section of a switch was found to scale linearly, so that with eleven pairs of processors communicating simultaneously, the bandwidth was one GB/s, Figure 2. The significance of this result is that system behavior is shown to be the sum of the individual node-to-node communications. It is for this reason that this paper confines tests to node-to-node communication, and does not consider group communications. However, we accept that in some cases the distinguishing factor in group communication may not be link behavior but software implementation and refer the reader to [21] where these effects are analyzed.

2.1 Installed Software

The software versions considered are: openMosix [22] version of MOSIX [9, 11] (for the purposes of this paper, hereafter referred to as MOSIX); the MPICH-2 [23] implementation[5] of MPI-2 (hereafter referred to as MPI-2); and also directly on top of standard Unix communication mechanisms [24], as implemented in the Gentoo version of Linux[6]. The Linux kernel version for the tests was 2.4.22. The 'C' library version was 2.3.2 and the version of the GNU `gcc` compiler was 3.2.3 20030422. The following aggressive compiler settings were chosen in compilation of the Linux source code: `-O3 -march=athlon-xp -funroll-loops`

[5] Available from `http://www-unix.mcs.anl.gov/mpi/mpich2/`

[6] Gentoo Linux, available from `http://www.gentoo.org/`, supports automatic upgrades on a daily basis, which assists in software management of the Linux cluster.

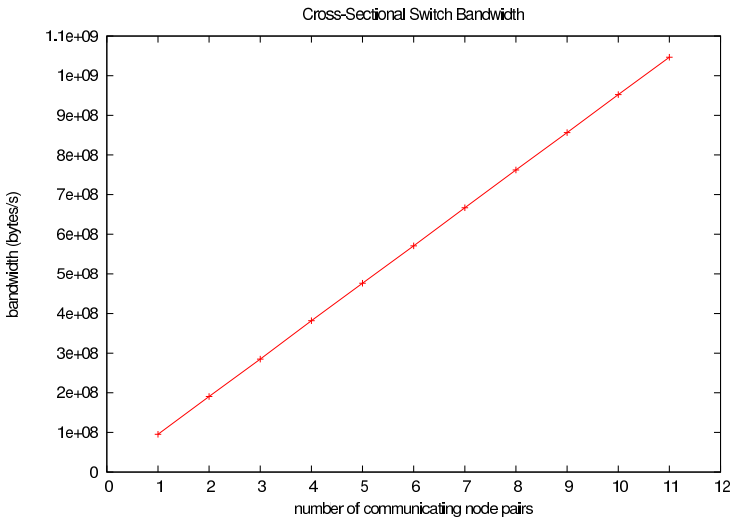

Figure 2: Scaling of cross-sectional bandwidth across an Ethernet switch

`-fprefetch-loop-arrays -pipe`.[7] As is clear, an Athlon XP specific setting was chosen. The loop unrolling setting selects those loops for which the number of iterations can be determined at compile time. The 'prefetch' setting requests memory pre-fetching for large arrays. Finally, `-pipe` only affects how `gcc` calls its backend utilities.

The relative performance of various MPI implementations was originally compared in [25], from which tests MPICH emerged favorably. The version of MPICH used in our tests was MPICH2-0.96p2. At the time of the paper's experiments, the license terms for MOSIX had apparently changed from those of the Gnu Public License, and it was reported that some sites had migrated to openMOSIX as a result. The vitality of the development for Linux was also an issue at the time of installation. This paper makes no special claims for either openMosix or MOSIX, and the authors are unconnected in any way to the implementors of openMosix or MOSIX. The implementation tests allow direct comparison of the performance of MOSIX and MPI-2, using the Unix mechanisms as a baseline.

Given that the CML model employs explicit incorporation of communication calls and process spawning, it was inappropriate to install a virtual shared memory system, though this installation would have allowed testing of the OpenMP standard [26].

MOSIX is a *preemptive process migration* system. As it runs in kernel space, MOSIX is transparent to applications; however a simple API exists that would allow applications to be aware of the cluster configuration and their location within it. Migration transparency is

[7] For details of speed optimization level 3 (the highest) and other settings refer to `http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Optimize-Options.html`.

achieved as follows. The process is separated into a user context (also known as 'the remote'), which can be located anywhere in the cluster, and a system context (also known as 'the deputy'), which remains on the home node. As a process must interact with its environment via its system context (in other words using kernel system calls), and the interface between system and user context is well defined, it is possible to capture this interaction and execute it at the present location of the user context (if the call is site-independent) or forward it over the network to the system context. This interaction is shown in Figure 3.

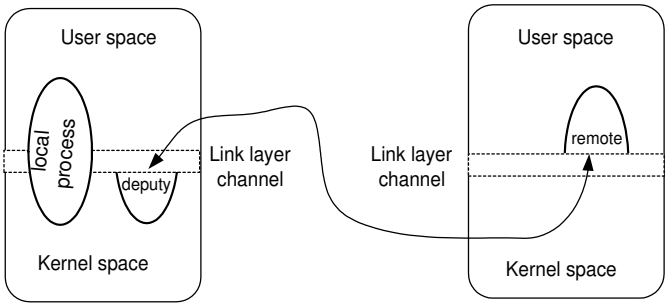

Figure 3: Local, and migrated MOSIX process, showing the data link layer channel for deputy to remote communication, after [8]

Instead of a spawn (creating a new process), MOSIX employs a Unix `fork()` to create a duplicate or child process. After the `fork`, the program counter of parent and child process is at the same position, and the two processes are only distinguished by the `fork` return value. In `fork`-style programming, there is a reliance on parent and child inheriting the same environment, rather than, as is possible, immediately 'execing' a new process.

3 Experiments undertaken

This Section describes the measurement experiments made. For convenience the Unix operating system communication mechanisms are summarized in Table 1. When not otherwise stated, Unix mechanisms normally derive from the BSD variant of Unix, in which data is normally passed as byte streams, whereas System V STREAMS [27](from research originating in [28]) derives from the AT&T variant of Unix, in which data is passed as discrete messages.

The base communication and spawning mechanisms in the experiments are either part of the Linux release or pre-written software packages (refer to Section 2.1 for their implementation). The Linux-based software is in widespread usage; it is open source and its design is widely reported in discussion forums. The experimental implementation code in its entirety

was written by the first author, with extensive software engineering experience, as noted in his biography below. A semantically well-defined wrapper layer was written around the software under test. This lightweight layer was consistently implemented, whether the software under test was MPI, MOSIX or one of the Unix mechanisms. Care was taken that no extraneous algorithms by way of the wrapper layer were introduced that would cloud the accuracy of the results.

Name	Description
`pipe`	Supplied with all flavors of Unix, communicating processes must have a common ancestor (after forking), normally half-duplex, byte streams
fifo (or named pipe)	Introduced in System III, identified by a path name, allows unrelated processes to communicate in half-duplex mode, byte streams
messages	System V variant, any permitted process can place a message on a queue, and any permitted process can remove a message, limits imposed on message sizes and numbers in queue, discrete messages with priorities
sockets	BSD origin, full duplex communication through network protocols, many variations[29]

Table 1: Communication mechanism glossary

3.1 Bandwidth and latency experiments

The benchmarking code spawns two processes, which communicate with one another. For the bandwidth test, one sub-process acts as sender and the other acts as receiver, Figure 4. For the latency test, a single byte is ping-ponged between the two processes 10,000 times,

undergoing an even number of transits so that a single clock can be used for round-trip time (rtt) measurement, also shown in Figure 4. The latency timings were observed to be variable but stability was achieved by averaging 10,000 rtt.

In general, the two processes are spawned on:

1. the same processor as the parent process
2. two different processors, neither of which is the parent processor
3. two different processors, one of which is the parent processor

Option (1) allows a simple comparison with the Unix mechanisms to be made, while options (2) & (3) investigate any bias that might result from the location of the parent processor.

In the MPI case, for the reason explained below, bandwidth and latency testing for option (2) was not possible.

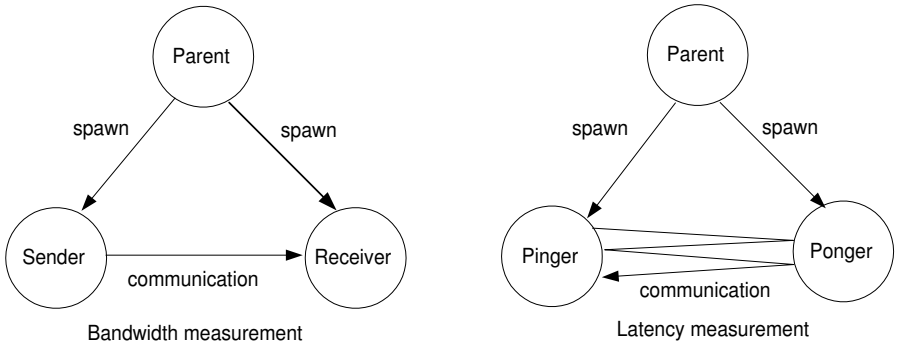

Figure 4: Communication test method

Table 2 gives the communication mechanisms that were examined. When standard Unix mechanisms [12, 13] that do not normally communicate between compute nodes are used in a MOSIX environment, they effectively do communicate between different nodes. When both nodes are not the home node, then the two MOSIX processes communicate via a home node, using the 'deputy' mechanism of Section 2.1.

Some communication mechanisms require the ends of the communication channel to be 'opened' centrally by the parent, whereas others require the child process to open the ends of the communication channel, and in other cases either possibility is allowed. For example, two processes talking via TCP sockets have to establish (open) the connection. This contrasts with a pipe operation, say, in which the two ends of the pipe are opened simultaneously by the parent process. In fact, this restriction inherent in socket communication considerably complicated the construction of the tests. The centralized channel opening of MPI-2 (called a

Mechanism	Inter-Node	Channel Ends Openable by Parent	Channel Ends Openable by Child
Unix pipe	No	Yes	No
Unix fifo (or named pipe)	No	Yes	Yes
Unix System V message passing	No	Yes	Yes
Unix TCP/IP sockets	Yes	No	Yes
MOSIX pipe	Yes	Yes	No
MOSIX fifo (or named pipe)	Yes	Yes	Yes
MOSIX System V message passing	Yes	Yes	Yes
MOSIX TCP/IP sockets	Yes	No	Yes
MPI	Yes	Yes[1]	Yes

Table 2: Different communication mechanisms

[1]— Only if communication occurs between parent and child.

communicator) normally only provides parent/child communication and not child/child communication. A recent change to MPI-2 does allow child/child communication, but in the form used by us, simultaneous spawning of two processes, the current MPICH-2 implementation apparently is flawed (see also Section 3.2). However, MPI-2 child-child communication is exactly the same as parent-child communication, unlike the MOSIX case, and, hence, nothing is lost by the omission.

3.2 Dynamic process creation experiments

When spawning a new process, there are two different strategies:

1. Allow the underlying system to make a suitable automatic choice of node — characterized by MPI-2 and MOSIX. MPI-2 generally allocates processes to its available nodes on a round-robin basis, whereas MOSIX attempts a more performance-informed allocation, only usually migrating processes from the home or starting node when this is considered good for overall performance. Indeed, the physical migration of a process may occur at any time.

2. Explicitly control the choice of target node— characterized by `RPC` and `rsh` calls.

To create a simple porting layer, a process placed version of spawn was used which straddles both strategies, *i.e.* it was easier to override MPI-2's and MOSIX's default behavior to achieve explicit placement than to emulate intelligent process placement of MPI-2 and MOSIX. In fact, MPI-2's and MOSIX's placement algorithms are incompatible, so there is no suitable unification mechanism. The main interest in MOSIX was the ability to achieve a transparent spawn (*i.e.* remote fork) between nodes, which was one reason why MOSIX's main feature, load-balancing, was disabled. Therefore, developer's should be aware of this limitation to our tests.

The efficiency of spawning was also measured. The spawning performance test methodology is shown in Figure 5: a process repeatedly spawns itself across two nodes to work out the cost of an individual spawn operation. The spawn operation has been implemented in seven different ways (refer forward to Table 4 for a list). The different types of spawning should be self explanatory from their names except:

RUNON is a MOSIX command, which can be used to start a remote process on a particular machine, just like `rsh`, or `ssh`.

MPI indicates the use of the dynamic process creation command added in MPI-2 [5] to MPI [30].

However, it was found that the current MPICH-2 implementation of MPI-2 only allowed random placement of processes, overriding directed placement. Moreover, it is entirely possible that the two processes will be placed on the same node, if spawned one after the other. MPI-2 does allow spawning of two or more identical processes at the same time and in this case the current MPICH-2 places the processes on different nodes.

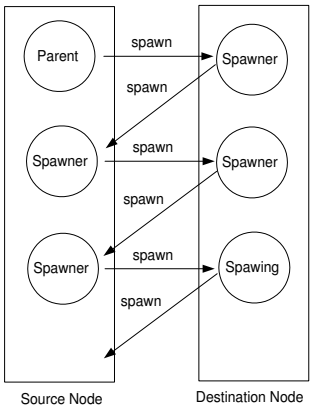

Figure 5: Spawning test method

The RPC implementation bundles *both* the automatically generated server and client code (produced by the Unix source-to-source code compiler `rpcgen` originating from SUN microsystems [13]) into the process executable. This is somewhat unusual, and some automated code re-writing is necessary to make this possible. So the RPC parallel application has one copy of a process on every node acting as server and one copy of the process is further needed on one particular node to initialize. Each RPC then forks a new copy of the application from one of the servers.

4 Results

4.1 Bandwidth and latency measurements

Figure 6 shows the bandwidth test results for the communication mechanisms of Section 3.1. Though there are small discernible differences between the Unix mechanisms, within a MOSIX context the bandwidth drops to a mere fraction - the MOSIX curves are essentially lying on the x-axis For intra-processor communication and message sizes in the range 10 to 100 bytes, socket communication was the most efficient, even though the socket Internet domain option was selected for inter-processor communication (and not the lightweight Unix domain). For larger messages sizes, `pipes` and `fifos` [12, 13] became the most efficient mechanism with little to separate the two. All Unix mechanisms are approximately equivalent at 1 GB/s when message sizes are 10,000 bytes or more.

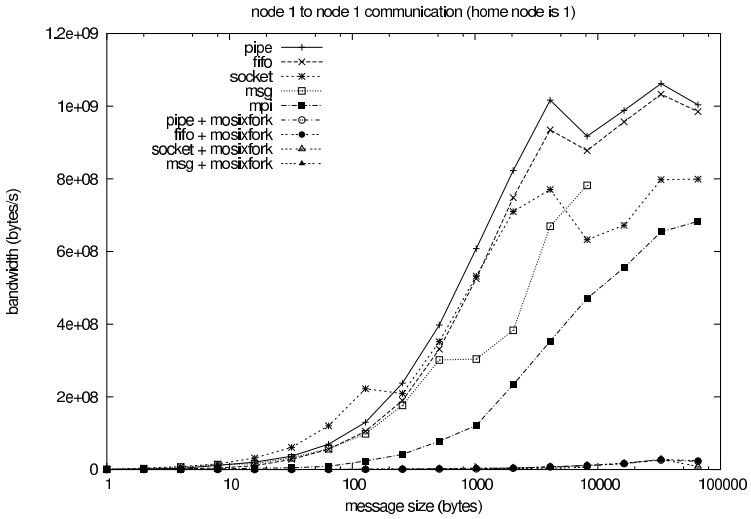

Figure 6: Communication bandwidth: node 1 to node 1

System V messages (`msg` [12, 13] in Figure 6) performed least well, though not all System V message sizes were tried, as superuser permission is required to change the maximum System V message size. There is also an upper limit on user designated socket communication message size. Socket communication is different as, in any single operation, the number of bytes requested is not necessarily sent (or received). and, therefore, it was necessary to 'code around' this limitation. Although a total message size is supplied, any particular communication may occur as a number of smaller messages (which in turn are broken up into Ethernet frames).

Figure 7 shows external bandwidths across a Gb link. Because of the MOSIX deputy mechanism of Section 2.1 all MOSIX communication in Figure 7 passes via home node 1 (the so-called MOSIX Universal Home Node or UHN) . As a result, it is clear that all MOSIX communication is sub-optimal, its effective bandwidth falling far below the bandwidth achieved by simple socket communication. Socket communication bandwidth levels out at around 100 MB/s, which is 80% of the available bandwidth, indicating good efficiency.

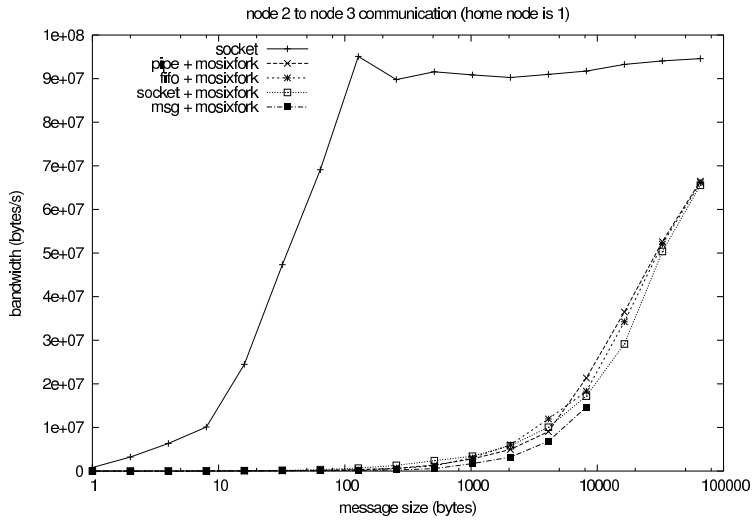

Figure 7: Communication bandwidth: node 2 to node 3

In Figure 8, the MOSIX bandwidth performance improved when the sender node is the UHN. However, surprisingly, even though there is no physical indirection through a parent node, the effective MOSIX bandwidth still falls far below that of socket communication. This is clearly a disappointing result and, in the view of the authors of this paper, requires attention by the MOSIX implementors. MPI performance falls firmly between the two camps, exhibiting an interesting plateau in performance between 100 B and 10 KB. We suspect that this plateau may be due to a particular buffer size in use within this version of MPI. When the message size

exceeds a threshold, then a different (and more efficient) mechanism may take over. Table 3

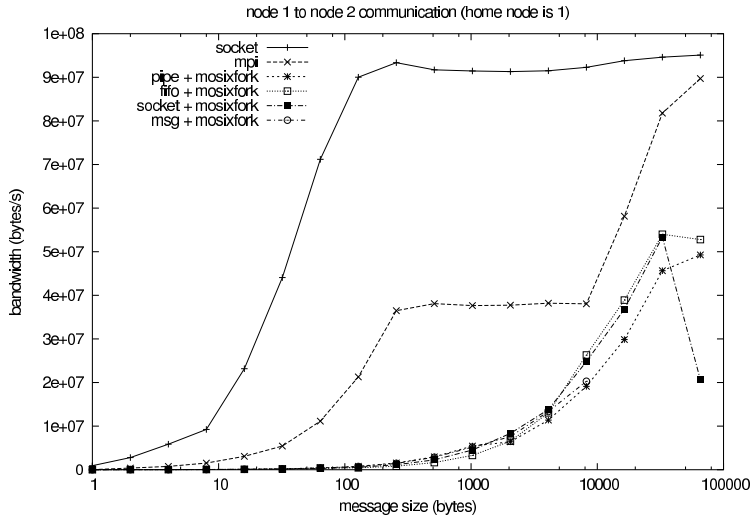

Figure 8: Communication bandwidth: node 1 to node 2

shows that, when using standard Unix Inter-Process Communication (IPC) mechanisms, such as `pipes`, `fifos`, and System V `messages`, latency is about 2 μs. Such communication is only possible between processes on the same processor. The latency resulting from Internet domain socket communication is about 7 μs on the same processor. Shaded boxes in the table indicate that the latency figure has been derived with MOSIX in operation. On a single processor the effect of MOSIX on latency is minimal (unlike the dramatic effect on bandwidth shown earlier); only the single processor SOCKET latency performance with/without MOSIX is given and the overhead is observably small.

A marked latency degradation was encountered between processors. The latency of IPC mechanisms operating between processors under MOSIX increases to around 350 μs, a couple of orders of magnitude worse. Additionally, inter-processor System V message queue latency is still larger: 550 μs.

Socket and MPI communications have a latency of around 240 μs, the lowest of all the methods measured. Socket communication via the MOSIX kernel increases latency by about 50%. The relatively small variations suggest that much of the latency can be ascribed to unavoidable features of Gb Ethernet technology, *e.g.* packet aggregation in order to reduce interrupt frequency, and packet delay in switches.

Communication Latency on Workstation Cluster			
Communication Mechanism	Communication route (node src → node dst)		
	1 →1	2 → 3	1 → 2
PIPE	1.95 ± 0.07 μs	360 ± 10 μs	330 ± 10 μs
FIFO	1.99 ± 0.03 μs	360 ± 10 μs	360 ± 20 μs
MSG	2.02 ± 0.05 μs	550 ± 10 μs	540 ± 20 μs
SOCKET	7.6 ± 0.2 μs	330 ± 20 μs	337 ± 3 μs
SOCKET	6.9 ± 0.1 μs	240 ± 20 μs	229.0 ± 0.9 μs
MPI	14.1 ± 0.2 μs	-	238 ± 9 μs

Table 3: Communication latency results

4.2 Dynamic creation measurements

This section benchmarks the time to "spawn" a new process using a variety of different mechanisms. The timings on the Linux cluster are given in Table 4. Additionally, timings on a 1.8 GHz Athlon XP 2200+ based PC (256 KB level 2 cache but the same main memory configuration as for the cluster processors) are given in Table 5: these values serve principally to show the SSH timings. SSH was not installed on the cluster as internal security was unnecessary for our activities. However, it was interesting to note from the PC test that there was no apparent performance 'hit' due to the addition of a security layer to RSH.

Spawning on the Linux cluster			
Mechanism	Time (node src → node dst)		
	1 → 1	2 → 3	1 → 2
FORK	61 ± 4 μs	N/A	N/A
MOSIX FORK	260 ± 10 μs	40200 ± 400 μs	14310 ± 70 μs
RSH	7550 ± 80 μs	68600 ± 400 μs	78800 ± 500 μs
RUNON	64600 ± 500 μs	70100 ± 300 μs	80600 ± 300 μs
RPC	1590 ± 40 μs	4300 ± 700 μs	4500 ± 400 μs
MPI	25703 ± 700 μs	-	29500 ± 100 μs

Table 4: Inter-processor spawning performance on the Linux cluster

MOSIX FORK is roughly as efficient as a standard FORK when source and destination

node are one and the same. Turning to other spawning mechanisms, these can be ranked as follows:

$$MOSIXFORK \gg MPI \quad \text{when} \quad src = dst$$
$$MOSIXFORK < MPI \quad \text{when} \quad UHN \notin \{src, dst\}$$
$$MOSIXFORK > MPI \quad \text{when} \quad UHN \in \{src, dst\}$$

Use of the RSH, SSH or RUNON implementations is not recommended by us, as a single spawn takes typically as much as one tenth of a second. RPC is very efficient on a Gigabit Ethernet network — there is much less network hit than with anything else.

Spawning on a 1.8 GHz Athlon XP based PC				
Mechanism	Time			
FORK	150	$\pm$	8	μs
MOSIX FORK	-			
RSH	177000	$\pm$	5000	μs
SSH	176000	$\pm$	5000	μs
RUNON	-			
RPC	1070	$\pm$	70	μs
MPI	22700	$\pm$	400	μs

Table 5: Intra-processor spawning performance on a PC running Linux

Table 6 is a "top six" of remote spawning mechanisms for the implementations measured on the Linux cluster. Averaging the MOSIX mechanisms timings, which differ according to the location of the UHN, allows another simpler form of ranking, as recorded in Table 7. The 'index' column in Tables 6 and 7 is a simple way of presenting relative performance. This crude metric is appropriate, given the wide range of performances. The fastest case has an index of one. A case with an index of five is approximately five times as slow and so on.

5 Conclusions

The results of this paper expose a ranking in terms of communication software performance. They reveal poor performance in certain circumstances, well below the hardware specification, which it is as well that the developer is aware of. The performance results indicate that MPI-

Top Six Distributed Spawning Performance				
Position	Mechanism	Condition	Time	Index
1.	RPC		4400 μs	1
2.	MOSIX FORK	$UHN \in \{src, dst\}$	14310 μs	3
3.	MPI		29500 μs	7
4.	MOSIX FORK	$UHN \notin \{src, dst\}$	40200 μs	9
5.	RSH/SSH		68600 μs	15
6.	RUNON		70100 μs	16

Table 6: Ranking of spawning mechanisms for implementations on the test machines

Top Three Distributed Spawning Performance			
Position	Mechanism	Time	Index
1.	RPC	4400 μs	1
2.	MOSIX FORK/MPI	29500 μs	7
3.	RSH/SSH/RUNON	70100 μs	16

Table 7: Reduced ranking of spawning mechanisms for implementations on the test machines

2 and MOSIX are broadly comparable in respect to spawning, and communication latency. However, the communication bandwidth under MOSIX is low, and optimizations are required if MOSIX were to become suitable for real-time or high-bandwidth applications. This paper has *not* presented results for MOSIX load-balancing performance, as our purpose was to take advantage of MOSIX spawn transparency. Lower-level communication mechanisms significantly outperform that provided by the high-level software packages tested by us, questioning gains offered by convenience or portability.

In general terms, the paper establishes that choice of transport software on a cluster remains a vitally important decision. Whatever, the advantages of a particular model of parallel computing, ultimately parallel computing is judged on its ability to deliver performance. However, with greater awareness of the importance of transport software, then implementors will be more likely to improve critical bottlenecks in performance. A logical further step is to investigate which transport software architectures lead to better communication and spawning performance, by investigation of the underlying structures, such as buffering, software layering, and interaction with the operating system kernel.

Acknowledgements

Dr Adrian Clark is thanked for the opportunity to run these tests on the Heap Linux cluster.

Stelios Bounanos is thanked for his effective adminstration of the cluster and its software.

References

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT, Cambridge, MA, 1994.

[2] C. Lin and L. Snyder. ZPL: An array sublanguage. In *6th International Workshop on Languages and Compilers for Parallel Computing*, pages 96–114, 1993.

[3] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, UK, 1999.

[4] D. J. Johnston, M. Fleury, and A. C. Downton. Prototyping Application Models in Concurrent ML. In *Euro-Par 2003 Parallel Processing*, pages 750–759. Springer, Berlin, 2003.

[5] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT, Cambridge, MA, 1999.

[6] T. Sterling, editor. *Beowulf Cluster Computing with Linux*. MIT, Cambridge, MA, 2002.

[7] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: a comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[8] A. Barak, O. La'dan, and A. Shiloh. Scalable cluster computer with MOSIX on LINUX. In *Linux Expo'99*, pages 95–100, 1999.

[9] Barak A., Guday S., and Wheeler R. *The MOSIX Distributed Operating System, Load Balancing for UNIX*. Springer-Verlag, Berlin, 1993.

[10] M. A. Baker, G. C. Fox, and H. W. Yau. A review of commercial and research cluster management software packages. *NHSE Review Electronic Journal,* at `http://nhse.cs.rice.edu/NHSEreview/96-1.html`, 1(1), 1996.

[11] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[12] R. J. Leach. *Advanced Topics in UNIX*. Wiley, New York, 1994.

[13] W. R. Stevens. *UNIX Network Programming*, volume 2: Interprocess Communication. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1999.

[14] R. W. Hockney. *The Science of Computer Benchmarking*. SIAM, Philadelphia, PA, 1996.

[15] I. Pyarali, T. H. Harrison, and D. C. Schmidt. Design and performance of an object-oriented framework for high-speed electronic medical imaging. *USENIX Computing Systems*, 9(3):265–298, 1996.

[16] D. C. Schmidt and T. Suda. Transport system architectures for high-performance communication systems. *IEEE Journal on Selected Areas in Communication*, 11(4):489–506, 1993.

[17] T. Sterling. Node hardware. In *Beowulf Cluster Computing with Linux*, pages 31–60. MIT, Cambridge, MA, 2002.

[18] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-PCs. In *IEEE Aerospace*, volume 2, pages 79–91, 1997.

[19] R. Breyer and S. Riley. *Switched, Fast, and Gigabit Ethernet*. Macmillan, San Francisco, CA, 1999.

[20] T. Sterling. Network hardware. In *Beowulf Cluster Computing with Linux*, pages 113–130. MIT, Cambridge, MA, 2002.

[21] T. H. Dunigan jr., J. S. Vetter, J. B. White III, and P. H. Worley. Performance evaluation of the Cray XI distributed shared-memory architecture. *IEEE Micro*, 25(1):30–40, 2005.

[22] M. Bar. openMOSIX, an open source Linux cluster project, at `http://www.openmosix.org/`, 2002.

[23] D. Ashton, W. Gropp, E. Lusk, R. Ross, and B. Ronen. MPICH2 design document. Technical report, Argonne National Laboratory, 2003. Report # ANL/MCS-TM-00.

[24] W. R. Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1998.

[25] N. Nupairoj and L. Ni. Performance evaluation of some MPI implementations on workstation clusters. In *Scalable Parallel Libraries Conference*, pages 98–105, 1994.

[26] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA, 2001.

[27] J. Peacock. Gently down the STREAMS. *UNIX Review*, 9:33–38, 1992.

[28] D. Ritchie. A stream input-output system. *AT&T Bell Labs Technical Journal*, 63:311–324, 1984.

[29] W. R. Stevens. *UNIX Network Programming*, volume 2: Sockets and XTI. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1999.

[30] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT, Cambridge, MA, 2nd edition, 1998.

Keywords

Linux cluster, Software benchmarking, MOSIX

Footnotes

*The authors are affiliated to the University of Essex, Department of Electronic Systems Engineering, Wivenhoe Park, Colchester, CO4 3SQ, United Kingdom, tel: +44 - 1206 - 872817, fax: +44 - 1206 - 872900, e-mail: {djjohn,fleum,acd}@essex.ac.uk

1. Prior versions of MOSIX [9] were written for other versions of the Unix operatings system.
2. The abbreviation 'b' for bits is used throughout this paper.
3. The abbreviation 'B' for bytes is used throughout this paper.
4. In fact, RPC is usually layered on top of the socket API.
5. Available from `http://www-unix.mcs.anl.gov/mpi/mpich2/`
6. Gentoo Linux, available from `http://www.gentoo.org/`, supports automatic upgrades on a daily basis, which assists in software management of the Linux cluster.
7. For details of speed optimization level 3 (the highest) and other settings refer to `http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Optimize-Options.html`.

Biographies

David J. Johnston has worked as software engineer in research and development for 20 years, at ICL Ltd. and the Rutherford-Appleton Laboratory, UK. His strengths lie in generating and realizing algorithms for complex systems. His interests include languages and methodologies to shorten the software development process. He has recently completed a PhD at the University of Essex, UK in position identification for augmented reality. He has co-authored a book on Computer Graphics.

Martin Fleury is a Senior Lecturer at the University of Essex, UK, where he was also awarded a PhD in Parallel Image Processing. His first degree was from Oxford University, and he holds an MSc in Astrophysics from the University of London. He is the co-author of a book on parallel computing for embedded systems. He has authored thirty journal papers in the last ten years on parallel image and vision processing, performance prediction, real-time systems, reconfigurable computing, software engineering, and video and document compression.

Michael Lincoln has completed an MSc and PhD at the University of Essex, UK in the field of face recognition and face tracking. He is a Senior Research Officer in a project concerned with radar control of aircraft landings. The cluster mentioned in the paper was constructed, configured, and commissioned by Michael.

Andrew C. Downton was educated at Southampton University, UK, where he obtained a first class honours degree in Electronic Engineering in 1974, and a PhD in 1982, and where he was also a lecturer. In 1995 he was promoted to a personal Chair at the University of Essex, UK, and since 1999 he has been Head of the Department of Electronic Systems Engineering at Essex. His research interests include pattern recognition and image analysis; parallel computer architectures; hardware-software co-design; handwriting recognition; and document analysis. He is a Chartered Engineer and Fellow of the Institution of Electrical Engineers (IEE) and a Senior Member of the IEEE.

Corresponding author details

Dr. Martin Fleury
University of Essex
Electronic Systems Engineering Department
Wivenhoe Park
Colchester CO4 3SQ
Essex
United Kingdom

Tel.: +44 1206 872817
Fax.: +44 1206 872900 (marked for the attention of M. Fleury)
e-mail: fleum@essex.ac.uk