Linux Cluster Communication Software Performance Evaluation for Scientific Problems

S. Bounanos and M. Fleury
University of Essex, Electronic Systems Engineering Department
Colchester, Essex, CO4 3SQ, United Kingdom
email: {sbouna,fleum}@essex.ac.uk

Abstract

A Linux cluster with Gigabit Ethernet interconnect is a local and accessible resource for solving scientific problems, including finite-element simulations. As general-purpose protocol stacks are not designed for parallel computing, the delivered throughput and latency may be significantly below that suggested by the hardware specification of the interconnect. The paper evaluates communication software across the protocol stack from high-level communication harnesses, such as MPI, Charm++ and MOSIX, through intermediate communication primitives, such as sockets, streams, and pipes, to underlying protocols such as TCP and TIPC, as well as low-level User-level Network Interfaces such as GAMMA. The evaluation is not only in terms of short message latency and throughput but in terms of the efficiency of the solution, that is the throughput per computation overhead. There are a number of findings, such as the relative efficiency of TIPC with Open MPI; confirmation of the advantage of GAMMA with MPI; the weaker performance of the single system image software under test; and the enhancement from adding application-level load-balancing to the system-level load-balancing available in some communication harnesses.

1 Introduction

Linux clusters [1] offer a cost-effective solution to scientific research problems. A crucial decision in transferring scientific code to a Linux cluster is selection of communication software, as this will impact on factors such as the ease of development, portability, flexibility, and the efficiency of the solution. This work reports general issues weighed against performance benchmarks and experience across the protocol stack: at application layer seven of the well-known ISO Open Systems Interconnection (OSI) model [2], such as in the Message Passing Interface (MPI) [3], Open MPI (OMPI) [4], and Adaptive MPI (AMPI) [5]; at session layer five, especially in socket usage; and across underlying layers four, three and two, for which Transparent Inter Process Communication (TIPC) [6] promises improved short-message latency over the normal TCP/IP protocol suite. At data-link layer two, User-Level Network interfaces (ULNs) are a more direct way to improve latency and throughput.

As Gigabit (Gb) Ethernet is becoming a standard feature of Linux clusters, there is some concern that existing protocol stacks will not be able to cope and as a result ULNs have been an active area of research for some time.

At the application layer, though Parallel Virtual Machine (PVM) [7] and especially MPI have tended to dominate, other choices such as Charm++ [8], and MOSIX [9], which offer different programming models, are examined as these have an influence on the parallel algorithm employed. For example, MOSIX's single system image model favors centralized algorithms, as communication is constrained to pass through a central node. Such options offer load balancing at the system level but may either fail to deliver or bring attendant overheads, either through implementation failings or from the weakness of the basic concept. The communication software is evaluated by low-level metrics of performance, an adaptive simulation application for semiconductor modeling, and by a standardized set of application NAS benchmarks [10]. The cluster under test includes gigabit switching and high-performance AMD processors. In scaling up to thirty processors, it is of a moderate but accessible size.

The paper is organized as follows. Section 2 describes the cluster computer environment employed in the evaluation of software, to allow replication of any tests. This section will be of interest to those contemplating the management of a cluster for moderate scale research, as well as indicating hardware performance issues. Section 3 describes application-level communication harnesses, in which the emphasis is on alternative packages to the ubiquitous MPI. The same Section tests the performance of the packages in parallelizing scientific research code, which, in its sequential form, was a serious impediment to experimental turnaround time. Section 4 examines what advantage may arise from directly taking advantage of Unix communication utilities, as opposed to MPI messaging or to single system image software, which adds automatic load-balancing of processes linked by Unix inter-process communication (IPC). Performance in this Section is measured by low-level benchmarks. Continuing the descent through the protocol layers, Section 5 asks the question what if an alternative transport/network level protocol is substituted for the familiar TCP/IP suite. Performance in this instance is tested both by low-level benchmarks and within the framework of OMPI. At the data-link layer, alternative data-link protocols are possible, as is direct interaction with the network interface card. The latter raises obvious issues of portability. Performance in this Section is measured with the NAS application kernels. Finally, Section 7 summarizes some of the main points in respect to communication software performance.

2 Computing environment

The cluster employed consists of thirty-seven processing nodes connected with two Ethernet switches [11] amusedly designated the ”Heap”, cf. [12]. 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[1] and dual channel 1 GB DDR333 RAM. The Heap nodes each have an 82540EM Gb Ethernet Controller on an Intel PRO/1000 network interface card (NIC). This Ethernet controller allows the performance of the system to be improved by interrupt mitigation/moderation [13]. It is also possible to hardware offload routine IP packet processing to the NIC, especially Transmission Control Protocol (TCP) header Cyclic Redundancy Check (CRC) calculation and TCP segmentation. In TCP Segmentation Offload (TSO), the driver passes 64 kB packets to the NIC together with a descriptor of the Maximum Transmission Unit (MTU), 1500 B in the case of standard Ethernet. The NIC then breaks the 64 kB packet into MTU-sized payloads. A NIC communicates via Direct Memory Access (DMA) to the CPU, over a single-width 32-bit Peripheral Component Interface (PCI) running at 33 MHz in the case of the cluster nodes. Though both the single-width and double-width (64-bit at 66 MHz) standard PCI busses should cope with 1 Gb throughputs, in [14] for the GAMMA optimized ULN, the short-width PCI bus throughput was found to saturate at about 40 MB/s for message sizes beyond 2^{15}B. For the majority of the tests, the version of the Linux kernel was 2.6.16.11, and the compiler was gcc/g++ 4.0.4. In Section 4, earlier versions of the Linux kernel and compiler were used, as specified in the Section.

In Fig. 1, the nodes are connected via two 24 port Gb Ethernet switches manufactured by D-Link (model DGS-1024T). These switches are non-blocking and allow full-duplex Gb bandwidth between any pair of ports simultaneously. In [15], typical Ethernet switch latency is identified as between 3 μs and 7 μs, whereas generally more costly Myrinet [16] switch latency is 1 μs. 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 Fig. 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 unmanaged 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 [17].

To make maintenance and cluster-wide propagation of configuration changes easier, at 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) [18]. The development branch of Debian sid (unstable but more current) is used to manage upgrades to the Linux operating system. To (re)build nodes `tftp` and `udpcast` (both Linux utilities) are employed in a manner akin to a simplified `SystemImager` (software that automates Linux installs, software distribution, and production deployment, available from SourceForge). A customized script is available to synchronize packages and configuration with the cluster's master template whenever it is inconvenient to reboot and rebuild. This operation is simply accomplished through an `APT` package cache that is shared between nodes and a shared `debconf db` (a configuration database), and runs in parallel across the cluster by means of `dsh` (an implementation of a wrapper for executing multiple remote shells, such as the `rsh` or `remsh` or `ssh` commands). Nodes rarely need to be added or removed, though

[1]The abbreviations ’B’ for bytes and ’b’ for bits occur throughout this paper.

there is a script to do that too. The cluster is monitored with the scalable, distributed system `Ganglia` (refer to `http://ganglia.info`).

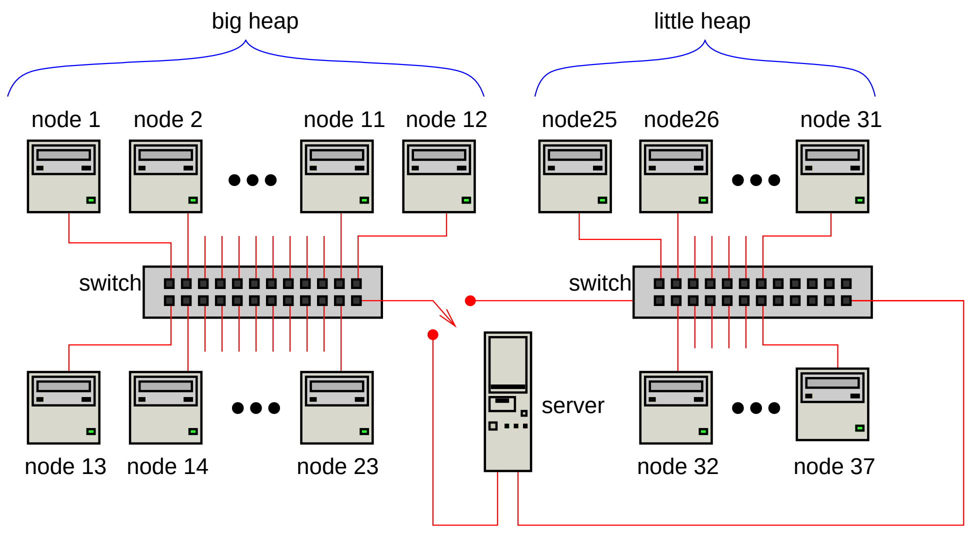


Figure 1: Schematic diagram of the Linux cluster, showing system partitioning and file-server

3 Application layer software

3.1 MOSIX

The MOSIX preemptive process migration system [9][19] originated as a symmetrical distributed operating system and on a cluster acts as a ’single system image’, performing automatic load balancing [20] by migrating processes to faster or idle nodes, or ’memory ushering’ [21] in which processes are migrated away from a node that is running out of memory to avoid heavy page swapping. Though MOSIX can be employed as a throughput engine, it has been recommended by its originators [22] for parallel applications on cluster computers, for example molecular dynamics simulations. The software version considered is the openMosix [23] version of MOSIX.

Migration transparency is achieved as follows. Referring to Fig. 2, 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.

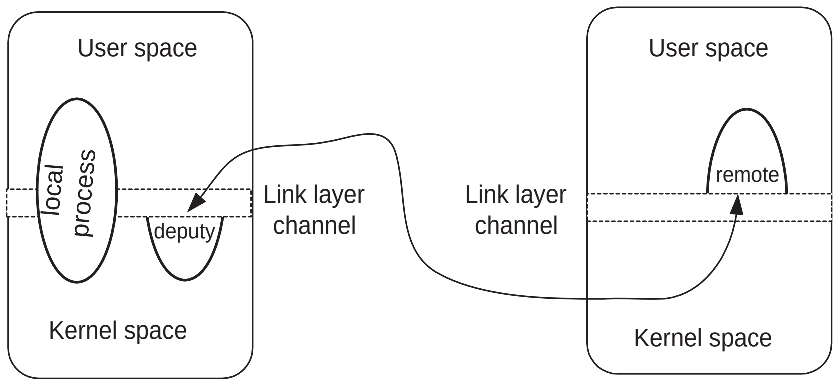


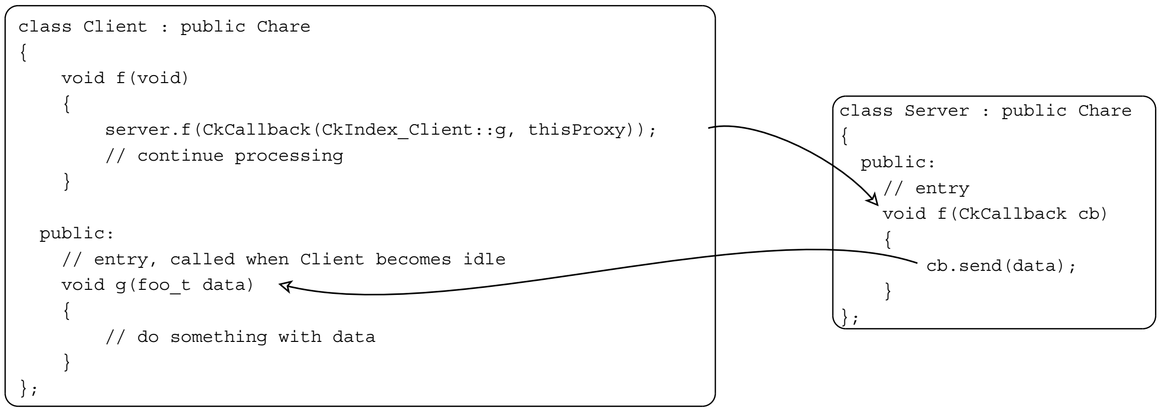
Figure 2: Local, and migrated MOSIX process, showing the data link layer channel for deputy to remote communication.

A MOSIX-enabled kernel employs the usual 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.

Given that under MOSIX file handles for filesystem and network based I/O are created on the home node and located in the system context, the separation imposes an overhead on communication. These issues are partially addressed by 'direct filesystem access' over a NFS-like filesystem known as the openMOSIX filesystem (oMFS) and ongoing work on 'migratable' network sockets, the aim being to reduce overhead by making some common I/O operations site independent. However, this did mean that migratable sockets were not available for this work.

3.2 Charm++

Charm++ [8] [24] is an object-oriented C++ development of the Charm parallel programming environment from the University of Illinois. Charm employs a message-driven programming style known as "split-phase" or "continuation passing". Fig. 3 gives a simple illustration of the continuation-passing model. In this model, computation and communication can overlap to a degree not attainable by other paradigms, *e.g.*, blocking remote procedure calls (RPCs), leading to a more "latency-tolerant" application. This does mean, however, that message queueing and scheduling are required at the system level, by linking in a runtime environment, and are possibly required at the application level. The Charm system is logically divided in two components, the Charm language [25] and the Charm runtime [26]. The Charm language, which conforms to the actor or active object model [27], is designed around:



```
class Client : public Chare
{
    void f(void)
    {
        server.f(CkCallback(CkIndex_Client::g, thisProxy));
        // continue processing
    }

  public:
    // entry, called when Client becomes idle
    void g(foo_t data)
    {
        // do something with data
    }
};
```

```
class Server : public Chare
{
  public:
    // entry
    void f(CkCallback cb)
    {
        cb.send(data);
    }
};
```

Figure 3: Continuation-passing in Charm++

- concurrent objects (called *chares*) which encapsulate data and computation. Remotely accessible chare functions are explicitly declared as such and become *entry points*;
- prioritized messages, the primary communication mechanism, which are delivered to remote entry points asynchronously and non-preemptively;
- data abstractions such as distributed tables, which are modified at runtime using messages, and read-only global variables which are defined at compile time and can be accessed synchronously, as they are copied to all instances of the Charm runtime.

To support data-driven execution, the Charm runtime has been structured into the following modules:

- The machine interface module which implements machine-specific functionality.
- The language features module that supports Charm's data abstractions and a number of common parallel programming algorithms such as quiescence detection.
- The system 'strategies' module which is responsible for load balancing and memory and message queue management.
- The core kernel which interfaces to the queueing module and runs a 'pick and process' loop to deliver messages to local chares or remote runtimes.

While Charm uses C as the base language, modules and chares are specified using non-standard syntax, requiring a pre-processing pass before compilation.

Charm++ has inherited the runtime architecture and retained most of the original language features, many of which have been rewritten to take advantage of C++ constructs. Two major additions to Charm are chare groups and chare arrays of arbitrary dimensions, intended for shared memory multiprocessors and networks of workstations respectively. Chare arrays are also the Charm++ collections of choice for operations such as barrier synchronization, reductions and message broadcasts. Fig. 4 shows interaction between chare array elements via message passing, from the programmer's and system's point of view.

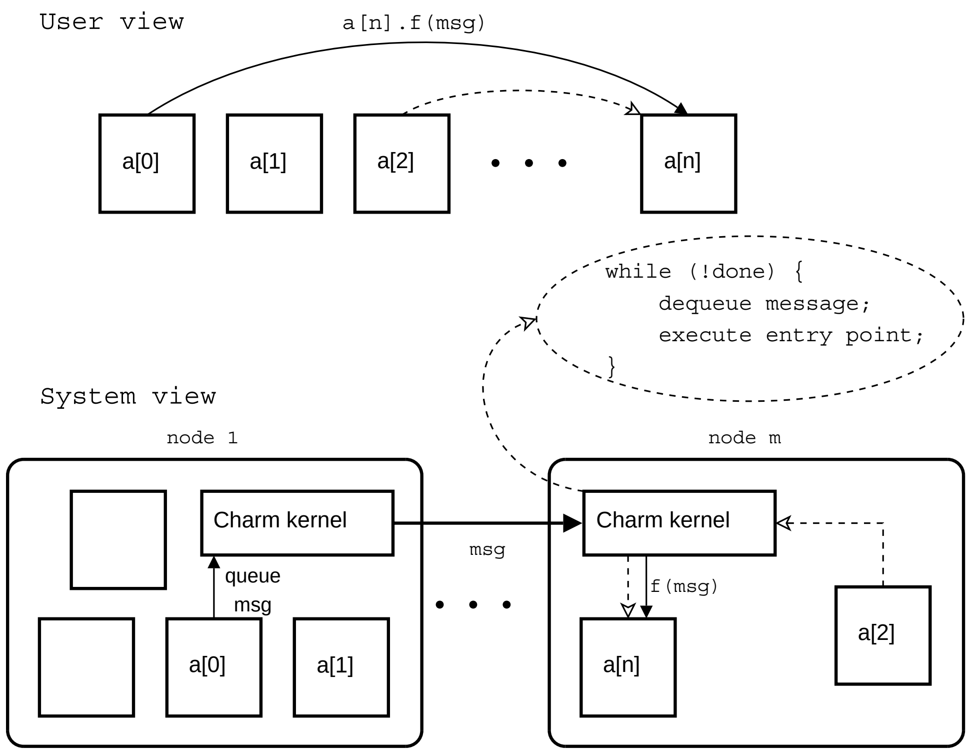


Figure 4: Charm++ system-user interaction

Charm++ provides measurement-based run-time load-balancing modules [28] for iterative applications, as these generally display a computation and communication pattern that persists over time. For applications with no or limited temporal behavior correlation, Charm++ employs 'seed' load balancers, which speculatively move seeds for new chares amongst processors. Dynamic load-balancing is only available to chares in chare arrays. All chares are instrumented but there is a choice as to whether statistics are collected centrally or in distributed fashion.

3.3 MPI: MPICH and LAM variants

The structure of MPI is very well-known and hence we simply refer the reader to one of the books on MPI [3][29]. For MPI-2 extensions, not considered herein, refer to [30]. Two principle variants or implementations of MPI were tested.

The LAM implementation of MPI [31][32] originally consisted only of the `lamd` Request Progression Interface (RPI) message-passing engine uses indirect communication among processes, via LAM daemons running on each node. As LAM is single threaded, this is currently the only RPI that provides truly asynchronous message passing. Subsequently, a variety of options have become available. The `sysv` RPI communicates by shared memory for MPI processes residing on the same node (with System V semaphores for locking), and by TCP to reach remote MPI tasks. A similar module, `usysv`, chooses spinlocks (with a backoff) rather than semaphores. `Usysv` is intended for Symmetric Multiprocessor (SMP) nodes with more processors than running MPI processes (of the same job), for which busy waiting can improve performance by avoiding preemption while the shared memory is locked.

When TCP sockets are used for message passing, LAM employs different application layer protocols depending on the message size. Short messages are sent immediately and in a non-blocking manner, while for other sizes a *rendezvous* protocol ensures that a matching receive call has been posted on the remote side before the message body is sent. For the same efficiency reasons, similar distinctions are made by the `sysv` RPI [33]. To optimize collective MPI communications, LAM also switches between linear and logarithmic algorithms depending on the size of the process group [34].

The MPICH implementation of MPI [35] is characterized by its wide range of native implementations, which it supports through its 'abstract device interface'. For example, like LAM MPICH has a shared-memory interface [36]. MPICH has three different protocols depending on message length. The default 'short' message size is 1 KB, which is small enough to be sent as part of a control message. 'Eager' messages of default length below 12 KB are sent immediately, whereas larger 'rendezvous' messages require a receive to be posted.

In [37], benchmarking on a workstation cluster confirmed that MPICH's large message threshold is larger than LAM's. Various performance studies, *e.g.* [38], [39], [37], give differing views of LAM and MPICH's relative performance, which varies by whether a custom parallel processor, (heterogeneous) network of workstations, or Linux cluster is employed.

3.4 AMPI

AMPI [5][40] is an implementation of MPI v. 1.1 on top of the Charm++ system. AMPI was introduced to make it possible for existing parallel applications to take advantage of Charm++ features without an extensive rewrite. Load balancing, migration and checkpointing calls are exported as MPI extensions. To support these, the main functions of MPI processes are renamed and run as user-level threads. All MPI peer and collective communication calls are mapped to wrappers and macros that in turn use the Charm++ runtime. This allows MPI processes to be migrated between Charm++ runtime instances as needed. Converted code must be made thread safe, principally by making global variables private (either manually or through the object code Execution and Linking Format (ELF) for position-independent code). Code that modifies global system structures, such as standard stream redirection for logging, also had to be adapted in an implementation.

Additional modifications are necessary to make MPI processes ‘migratable’. These are mainly a set of pack/unpack routines, which are registered with the Charm++ runtime and take care of the usual tasks of freeing heap-allocated memory and saving and restoring structures. These can also be used for checkpointing an MPI process by writing a serialized copy of its data on disk. Migration itself is apparently controlled by barrier synchronization, which is initiated by carefully placed `MPI_Migrate` calls.

3.5 Application benchmarking

To benchmark the communication harnesses a finite-element application was selected, as this was frequently employed in our institution to model novel optoelectronic semi-conductor devices. In [37], a similar application was chosen to benchmark the MPI variants, whereas a complex but rapidly converging finite-element algorithm, multi-grid, appears as one of the NAS application kernels [10]. The simulated drift-diffusion transport model [41] is applicable to a wide range of layered semi-conductor devices, whereas other models are limited to particular device geometries and bias conditions. The drift-diffusion transport model requires solution of a non-linear Poisson equation and current continuity equations in the classical domain, together with a numerical solution to Schrödinger’s equation if extending to the quantum domain. A method of decoupling the computation of the constituent equations using block iteration was devised by Gummel [42]. This method has a number of important advantages: it has global convergence in that convergence is guaranteed for any initial guess; the solution of simultaneous equations is avoided by updating the electrostatic and quasi-Fermi potentials at each mesh point by an explicit formula; and it only requires storage of the results of one previous iteration.

Parallel decomposition of the MOSIX version of the simulation initially took advantage of a slab approach. The grid was divided into segments of equal size and distributed to child processes. Communication in the MOSIX solver reflects the MOSIX architecture and is done in a centralized fashion. At the end of each iteration, the parent receives each child’s boundary rows as well as a value calculated from its local grid segment, the aggregate of which is checked against the convergence condition. Unless the algorithm has converged, the parent sends the boundaries to the owner’s neighbors to resume the computation. Message sizes before addition of protocol headers) were 1312 B, within the Ethernet physical frame size of 1538 B.

The Charm++ version employed a distributed algorithm, with chares sending boundaries directly to neighbors. In the initialization phase in the Charm++ version the global grid is declared as a read-only global variable and is initialized in the main chare’s constructor. The main chare then determines grid division and instantiates a chare array with the required number of elements, which are distributed to the pool of available nodes by the Charm++ runtime. At construction time, each chare creates its local grid segment from the global copy and, having completed its initialization, notifies the main chare that it is ready and becomes idle. When all chares are ready, the main chare broadcasts a message to the array to begin the computation. The same load-balancing strategy and settings were employed in AMP as in the Charm++ version.

The MPI version employed the same distributed algorithm as in the Charm++ version. The MPI processes use a combination of blocking and immediate calls [3] to maximize computation/communication overlap. At each iteration step:

1. The process blocks to complete an immediate receive request posted in the previous step.
2. An immediate, non-blocking, receive is posted for the next step, and the request stored.
3. The grid chunk with the new boundaries obtained from (1) is processed and the step number is incremented.
4. Boundaries for the current step are packed and sent to neighbors using a standard blocking send.
5. All processes perform a reduction and the result is checked for convergence.

3.6 Implementation details

As MOSIX processes can potentially migrate a number of times, it is important to keep the memory footprint of each child process as small as possible. To help reduce migration overhead, the parent process allocates the global grid and other initialization-only structures on the heap, where they can be freed by the children when no longer needed. When all children have connected, the parent locks itself to the home node using the MOSIX `/proc` API and triggers the computation. The children begin to process their segments, increasing the load on the home node and creating an imbalance in the cluster. This is detected by the MOSIX load-balancing subsystem and child processes are migrated away to idle or underloaded nodes within the first few hundred iterations (typically less than 1% of the total number). The MOSIX solver processes use Unix domain stream sockets. As MOSIX wraps all deputy-remote communication in its own User Datagram Protocol (UDP)-based protocol, the encapsulated protocol should be kept as simple as possible to maximize efficiency.

Of Charm++'s non-seed load balancers, the `RefineCommLB` module was selected for the application. Other balancers were experimented with but it was possible for the application to misbehave if a chare acted as a sink for a collective communication operation. In the application, a reduction collective operation collects results centrally after each iteration of the application. Therefore, only load-balancers for which a particular chare could be registered as non-migratable were suitable. Fortunately this includes the `RefineCommLB` module, which takes into account both communication and chare computational load, and, hence, was the most applicable of twelve available load-balancing strategies. Migration can be restricted to preset synchronization points, but the default option, periodic migration, was selected. Migration takes place unless the chare is completing computation of an entry point method. Centralized system measurement gathering was selected, incurring more overhead but in principle allowing more accurate decisions to be made. Boundary message sizes (before addition of protocol headers) were 632 B, but unlike MOSIX the boundary data is not aggregated. The total message data exchanged was 1264 B, similar to MOSIX. Though choice of UDP or TCP transport layer protocol is possible, UDP was chosen to reduce latency.

The simulation was also ported to two implementations of MPI, MPICH v. 1.2.5.3 and LAM v. 7.1.1. MPICH was compiled and run with `ch_p4mpd` support for job startup, while LAM's boot method was by `rsh`. In the LAM version of the simulation, as potentially many MPI processes run on the same cluster node, TCP is clearly inefficient as it does not take advantage of shared memory. The `usysv` RPI would cause processes to compete for CPU time even while not doing useful work. In preliminary investigations, `lamd` gave a net drop in performance due to its increased message passing latency. Therefore, the default `sysv` RPI was the optimal communication method. In the application, the message body size is always 624 B for boundaries (and much smaller for reductions), which is well below the short message threshold of all LAM RPIs. The boundary messages also fell below MPICH's short message threshold.

3.7 Performance

Fig. 5 compares the speed-up performance of the simulation with MPI and Charm++ without system load-balancing to Charm++, MOSIX, and AMPI, all with system-level load-balancing. The plot with legend 'charm' is a 'plain-vanilla' version of Charm++ without system load-balancing. The legend 'charm-rc' denotes Charm++ running with the `RefineCommLB` load-balancing module. The legend 'charm-sub' is a version of the application algorithm that synchronized on sub-cycles within an iteration. Though sub-cycles were included in the sequential version of the algorithm (and did not effect the performance of that version) the need to synchronize clearly reduced performance considerably, without apparently altering the end results. The vertical line in Fig. 5, at 37 nodes marks the exhaustion of physical nodes on the cluster. Thereafter, virtual nodes are employed, *i.e.* physical nodes hosting more than one process.

In Fig. 5, speed-up curves show almost linear speed-up for Charm++ up to the number of physical nodes. The workload is not evenly spread between nodes and as a result synchronization delays occur. When more than one (virtual) node is placed on a processor then, while one node completes its work, the other(s) can receive a message and continue the next step's processing. In the Charm++ distributed algorithm this results in continued speed-up beyond the number of physical nodes, provided the process placement is propitious. The MOSIX version cannot benefit from asynchronous operation because of the centralized communication pattern, resulting in a reduction in speed-up beyond the critical point. However, even before that point MOSIX's performance is disappointing in comparison in comparison to Charm++. Closer examination showed that the number of migrations in a run was considerable. In fact, the number of migrations rises significantly after the physical nodes are exhausted, and does not then always favor the load-balanced version.

The MPI-based systems behave very similarly to each other and to Charm++ until the number of physical nodes is exhausted. Given that AMPI uses the same load-balancing system as Charm++, the consistent drop in performance after 55 nodes indicates a weakness in the current implementation of the AMPI software. The addition of system load balancing to Charm++ does bring gains in performance if virtual nodes are available for migration. However, the erratic Charm++ performance for large numbers of nodes makes the circumstances when load-balancing is advantageous compared to an MPI implementation difficult to predict. There is little to choose between the two implementations of MPI. As in the Charm++ 'plain-vanilla', the MPI systems benefit from the reduction in latency from placing more than one node on a processor. Hence, speed-up continues to rise beyond the number of physical nodes. This can be ascribed to parallel slackness, allowing processing on one virtual node to proceed while another node on the same processor is temporarily stalled.

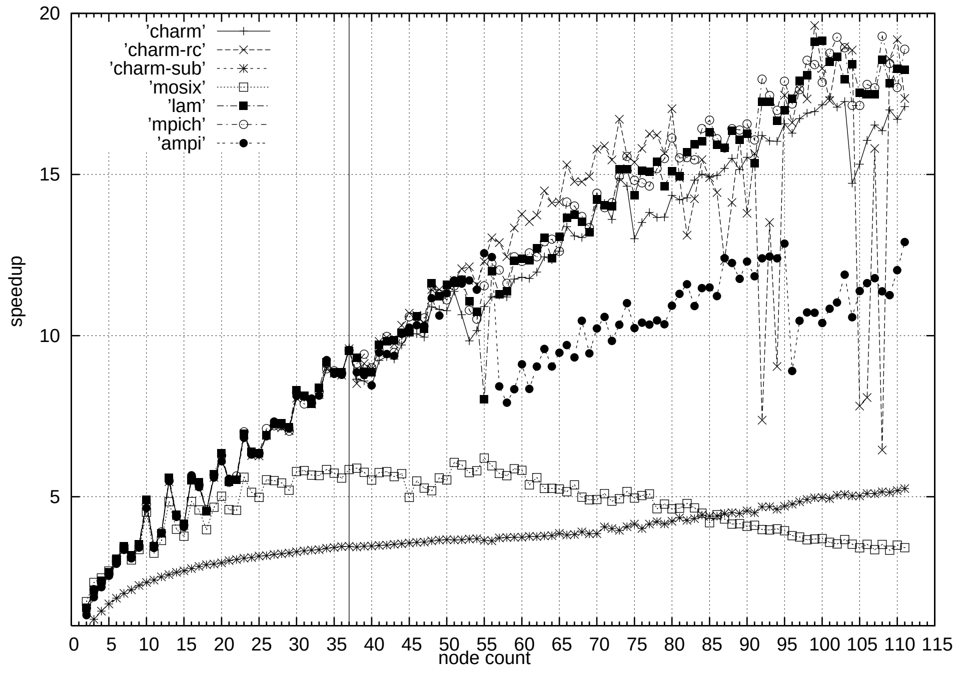


Figure 5: Speed-ups for load-balanced and non-load balanced systems

Unfortunately, equal division of the simulation grid results in severe load imbalance as there is a region within the grid which incurs steeply rising computation times during one particular step in the computation. Many processors become idle waiting for boundaries from those that have been allocated rows from that region. This was mitigated in part through finer granularity by increasing the number of grid segments and using virtual nodes (*i.e.* when more than one node is mapped to a physical processor or cluster node) to host the additional segments. Additionally, the code was instrumented, in the Charm++ case by the Charm++ Projections tool [43], and in the MOSIX version through the GRM library, which is part of the PROVE grid monitoring and visualization toolkit [44]. The results of this instrumentation allowed unequal-sized segment distribution, as a form of application-level load-balancing.

Application-level load balancing was added to all versions, with the results shown in Fig. 6. The MOSIX performance is essentially a scaled version of that without application-level load balancing. Charm++ and the MPI implementations were best able to take advantage of this type of load-balancing, though again AMPI's performance is weaker. At lower numbers of virtual nodes, the plot for Charm++ with system load-balancing (and application-level load-balancing) smooths out, making it easier to predict performance for a given configuration. Performance of the Charm++ and MPI versions gradually rises, as more and more virtual nodes are added. There is a thresholding affect as the number of nodes on any one processor exceeds two and then three. Adding both application load-balancing and system-level loading results in the best speed-ups. However, MPICH's performance without system load balancing is very competitive and exceeds that of the LAM implementation for large numbers of nodes.

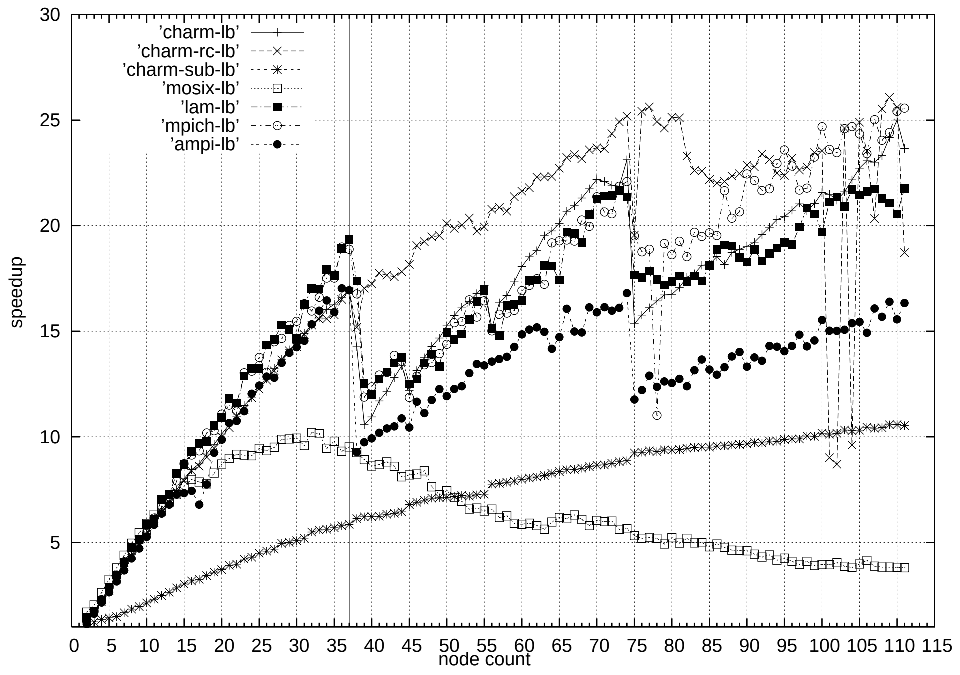


Figure 6: Speed-ups with application-level load-balancing for all systems

4 Session layer software

At the session layer there are a variety of communication mechanisms that can be employed, either directly, as part of single-system image software, such as MOSIX, or MPI can be deferred to.

The software versions compared in this Section are: again the openMosix [23] version of MOSIX [9, 19]; the MPICH-2 [45] implementation[2]; and standard Unix communication, as implemented in the Gentoo version of Linux. The Linux kernel version for these 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 -fprefetch-loop-arrays -pipe`.[3] 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.

For convenience, the Unix operating system communication mechanisms tested 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 [46](from research originating in [47]) derives from the AT&T variant of Unix, in which data is passed as discrete messages.

4.1 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, Fig. 7. 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 Fig. 7. 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.

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

[3]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`.

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[48]

Table 1: Communication mechanism glossary

Table 2 gives the communication mechanisms that were examined. When standard Unix mechanisms [49, 50] 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.

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 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 MPICH-2 implementation apparently was flawed. 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.

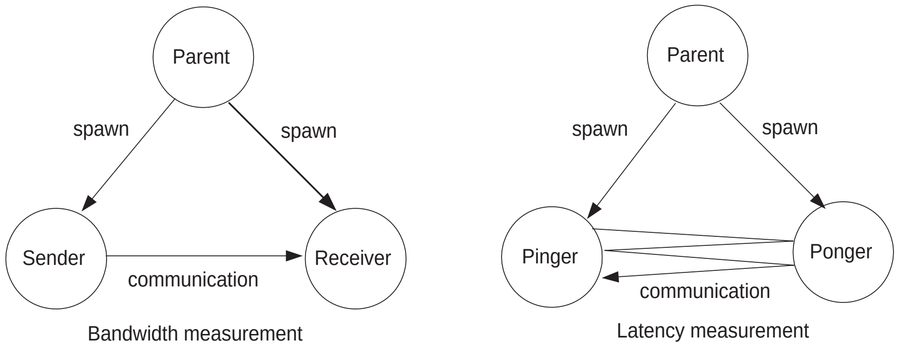


Figure 7: Communication test method

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.

4.2 Performance

Fig. 8 shows the bandwidth test results for the communication mechanisms of Section 4.1. 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 B, 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` [49, 50] 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 10000 B or more.

System V messages (`msg` [49, 50] in Figure 8) performed least well, though not all

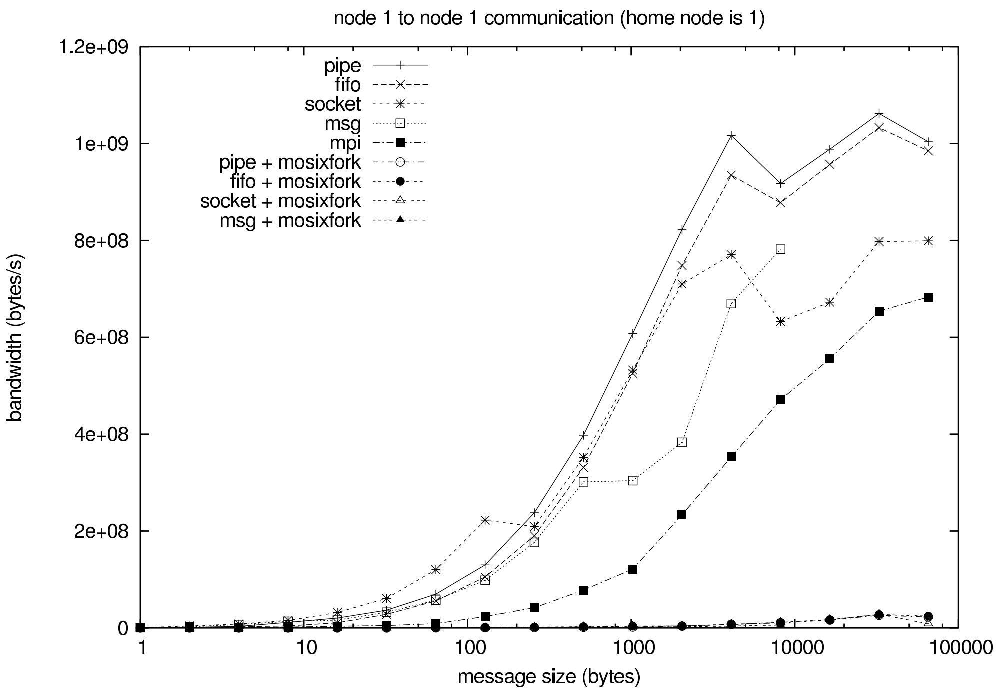


Figure 8: Communication bandwidth: node 1 to node 1 (Note the log. scale on the horizontal axis.)

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).

Fig. 9 shows external bandwidths across a Gb link. Because of the MOSIX deputy mechanism, all MOSIX communication in Fig. 9 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.

In Fig. 10, the MOSIX bandwidth performance improved when the sender node is the UHN. However, surprisingly, even though there is no physical indirection through a parent

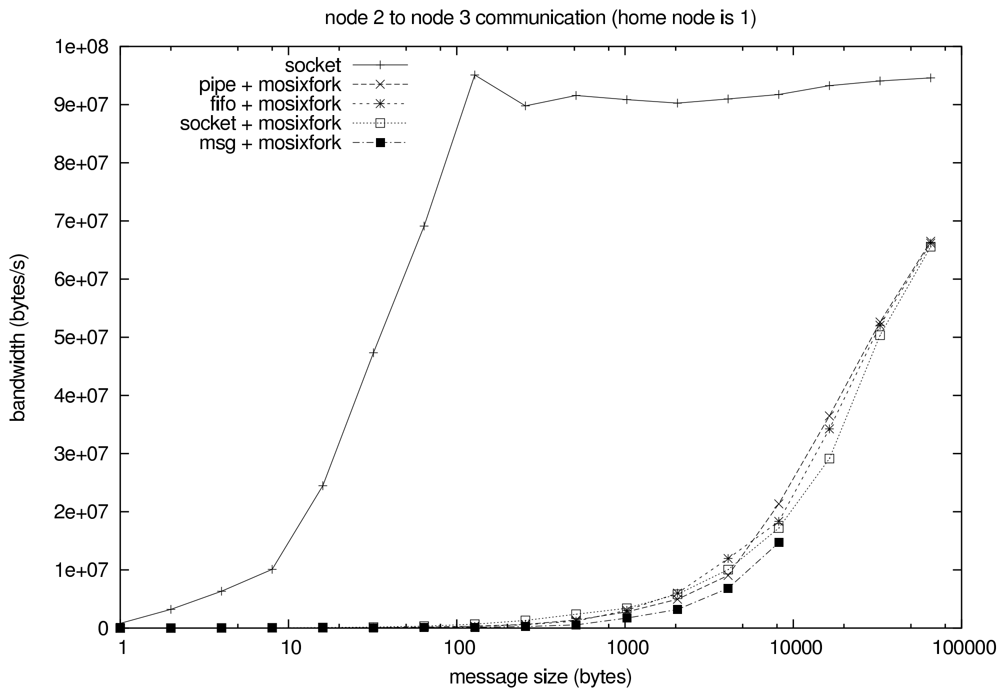


Figure 9: Communication bandwidth: node 2 to node 3 (Note the log. scale on the horizontal axis.)

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 shows that, when using standard Unix 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

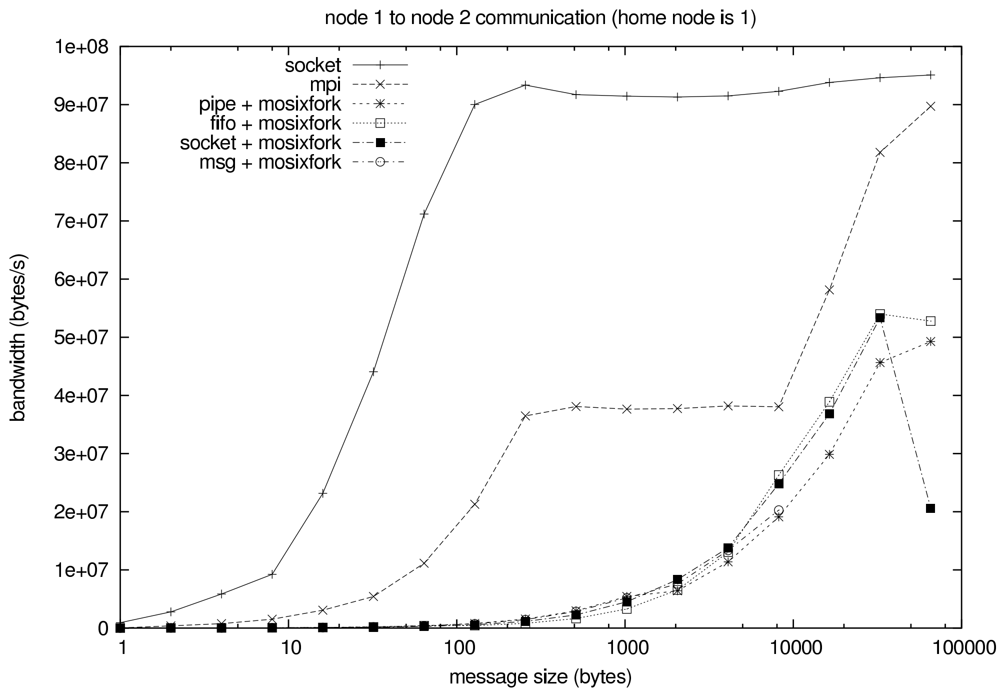


Figure 10: Communication bandwidth: node 1 to node 2 (Note the log. scale on the horizontal axis.)

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%.

5 Transport and Network layer protocol

This section reports possible advantages of employing an alternative transport and network layer protocol, Transparent Inter Process Communication (TIPC) [6][51]. TIPC is now widely available, being a feature of some Linux version 2.4 kernels and is now incorporated into Linux kernel v. 2.6. Conversely, the TCP/IP suite are the protocols of choice for generalized message-passing software because of their standardization in the Internet. TCP provides flow and congestion control, reliability, and limited error detection. UDP does not have the algorithmic overheads imposed by TCP, but it requires the parallel application

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

programmer on a cluster to guard against switch and processor buffer overflow, and, in general, check for message duplication or out-of-order delivery. IP provides multi-hop routing, relying for its addressing on the Domain Name System (DNS) service and for address translation on (Reverse) Address Resolution Protocol ((R)ARP). Local routing tables need to be set up if external look-up is avoided. ARP (or increasingly Dynamic Host Configuration Protocl (DHCP)) are based on local broadcast.

PVM/MPI have increasingly been deployed in closed cluster environments, rather than in a network of workstations (NOWs). A number of studies have identified the importance of latency to cluster applications [52][53], especially for short messages [54]. Reduced latency for small-sized messages and logical addressing are two of the attractions claimed by TIPC for clusters and, hence, for PVM/MPI. While file transfer is common in the Internet, in parallel computing on a cluster, short messages for synchronization, parameter updates, and transfer of border calculations occur frequently. Internet addressing is intended for dynamic routing, while cluster processing is about co-operation of processes, often only one hop away across an Gb Ethernet switch. Logical addressing allows real-time calculation of routes based on the zone (group of clusters), cluster, or sub-net within a cluster.

TIPC is adapted from Ericsson's TELORB IPC, with redesigned source code released as open source in 2004. TIPC fuses protocol layers [55], which reduces the number of memory-to-memory copies within the stack, a prime cause of increased latency in protocol stacks. In essence, the protocol stack can be written as a pipeline, while simultaneously exploiting data locality between processing steps. By means of signaling through a generic data-link layer, transparent message reliability is provided. Like TCP, TIPC imposes reliability on network links, which is present in (say) Myrinet but is absent in (say) ATM and Gb Ethernet. The latter two may drop delayed packets at ingress queues. Like TIPC, PANDA [56] also provides reliability whenever the underlying physical network is lacking. For example, both ATM and Gb Ethernet may drop delayed packets at incoming queues, while Myrinet's hardware back-pressure method [16] is reliable. In comparison to TIPC, PANDA [56] also provides reliability whenever the underlying physical network is lacking.

Connectionless and single message connection handshake remove the impediment of

TCP's three-way handshake when dynamic real-time messaging is required. TIPC employs a static sliding window for flow control rather than TCP's adaptive window, which avoids complex window size calculation algorithms. In case of link congestion, message bundling up to the MTU is practised. TIPC also delegates checksum error detection to the data-link layer, which without hardware-assist is a serious burden. For embedded systems, including multi-core processors, TIPC's small footprint ($\leq$ 20 k lines of code) is an added attraction. Compared to TCP/IP, it is also relatively easy to adapt to differing physical layer carriers.

5.1 Features of TIPC

Within Open MPI, the TIPC stack is accessed through the socket API and, thence, directly to an Ethernet frame or native data-link layer protocol. TIPC can also enlist another transport layer protocols as a message bearer, such as through UDP, shared memory in Wind River's real-time operating system VxWorks [57], the Stream Control Transport Protocol (SCTP) [58], or pass messages via mirrored cluster node memory [59]. However, the latter two bypass, some of the lower layers of the TIPC stack [6]. Specifically, the SCTP interface bypasses sequence control and mirrored memory bypasses packet fragmentation, message bundling and congestion control, as well as packet sequence control. There is also an ultra lightweight version of TIPC for intra-node communication, which removes the need to switch from TCP to UDP for same node communication.

An essential feature of TIPC is logical addressing, which unlike IP addressing, is independent of the network location of the communicating parties. TIPC addresses and their mappings to physical addresses are maintained in local name translation tables, which are updated by reliable multicast. More generally name lookup and configuration services operate transparently at the datalink layer. A TIPC address consists of a global port name, with a `type` field specifying the type of service supported by the port and the `instance` field acting as a key to a particular instance of that service. A `tipc_name_seq` alternatively specifies a range of addresses supported by a server, effectively multicast addressing. Alternatively, when logical addressing is inappropriate, a TIPC address may consist of a unique physical port (such as a `socket`) part of a `port_id`. In Linux, the TIPC address is mapped into a traditional `sockaddr_tipc` C structure for socket API access at an application.

5.2 Open MPI implementation

To compare TIPC and TCP/IP, TIPC was ported to Open MPI (OMPI) [4] and its component-based architecture. OMPI implements the full functionality of MPI-1.2 and MPI-2. It also merges the code bases of LAM/MPI [60], LA-MPI [61], and FT-MPI [62], capitalizing on experience gained in their development. With the growing scale of parallel processing, fault tolerance features [63] are also introduced to OMPI.[4] While OMPI is probably aimed

[4]In fact, TIPC also allows a process to subscribe to a notification service for processor or process crashes, and link failures. On link failure link re-routing is transparent to the application.

at terascale processing at national laboratories, its flexibility allows alternative protocols to be introduced. The component architecture makes it possible to provide drivers for diverse physical layers and diverse protocols. For example, Myrinet, Quadric's qsnet [64], Infiniband, and Gb Ethernet are alternative physical layers, while TCP/IP and now TIPC are alternative protocols. Alternative data link protocols are also supported such as Myrinet's Glen's Messages (GM). However, OMPI currently employs TCP for out-of-band (OOB) spawning of processes through the Open Runtime Environment (ORTE) [65]. For ease of protocol implementation, this feature may be retained (though implementations without TCP OOB are clearly possible [66]).

OMPI is organized as a series of layers of which the Byte Transfer Layer (BTL) is concerned with protocols. OMPI supports the normal MPI point-to-point semantics, namely standard, buffered, ready, and synchronous. However, OMPI's low-level library calls employ different types of internal protocols for point-to-point communications between OMPI modules. The type of internal protocol depends on message size. For some message sizes, OMPI supports software Remote DMA (RDMA) (or Remote Directed Message Access [67]). In hardware RDMA, messages are directly available (zero-copy) from/to the user application, by passing the operating system. Hardware RDMA is a feature of Cray XT series computers and the technology was transferred to clusters through Myrinet, together with Quadrics/Meiko's Elan4 NIC and others. The issue of operating by-pass for Gb Ethernet is taken up in Section 6.

In software RDMA, an initial message portion contains a message descriptor. If the receiver has already registered an interest in the message, then the remainder of the message is requested and transferred directly to user memory. Otherwise, the transfer only takes place when the receiver registers an interest and supplies the target user address. Function callbacks at sender and receiver in this rendezvous operation are used in the manner of Active Messages [68]. Compared to MPICH, OMPI therefore has an extra overhead from message matching but avoids the overhead from large unexpected large messages. An identifying message tag in the current implementation of OMPI is of size 16 B [66].

In summary, OMPI supports three internal protocols:

1. An eager protocol for small messages, up to 64 KB in the TCP BTL. If the receiver has not registered for a message, then there is an overhead in copying the message from the receive buffers to user memory.

2. A send (rendezvous) protocol with support for RDMA for messages up to the BTL's maximum send size; parameterized as 128 KB for TCP.

3. A pipeline protocol which schedules multiple RDMA operations up to a BTL-specific pipeline depth, for larger contiguous messages. In the pipeline, memory registration is overlapped with RDMA operations.

5.3 Performance

Fig. 11 considers the latency of TIPC against several different varieties of TCP. The comparison is made by normalizing the TCP latencies to that of TIPC. The TCP-BIC version of TCP [69] is the default setting of TCP in Linux kernel 2.6 and this was the version employed. Though TCP-BIC is specialized to networks with high bandwidth-delay products, it also has good all-round properties [70]. Comparative tests between TCP-BIC and TCP New Reno established that no appreciable difference occurred in the outcomes of the tests, and, hence, TCP-BIC was retained. In these tests and those of Section 6, the socket send and receive buffers were 103 KB in size, this being the default and immutable size for TIPC. The default value for the Gb Ethernet NIC transmit queue of 1000 (`txqueuelen`) was retained, whereas lower values are thought more appropriate for slower devices such as modems. It is possible [70] that increased throughput and/or increased acknowledgments may occur if `txqueuelen` is changed.

For throughput tests, a client process connects, sets socket options, sets an alarm timer for the test duration (30 s), starts its clock and enters a receive loop. Received data is counted by means of an unsigned 64 b counter. When the `ALRM` signal arrives, the client measures the actual test duration (with the Unix utility `gettimeofday()`) and calculates the average throughput as total_data / duration. The server continuously sends. `Gettimeofday()'s` resolution is more than adequate over 30s. For latency, the client repeats the above procedure, except that the client enters a recv/send loop and counts the number of completed recv/send pairs, N. The average one-sided latency is calculated as (duration / N) / 2. The server also runs a tight send/recv loop. When efficiency is calculated, a client calls the Unix utility `getrusage()` immediately after the test timer expires and finds the system CPU time (`ru_stime`) that has elapsed. Dividing throughput by system time is a measure of efficiency.

To establish basic behavior, tests were taken by running TCP and TIPC, as alternative protocols at the socket API level. In Fig. 11, the plot with the legend 'with hw offloading' refers to off-loading of the TCP/IP checksum calculations and TCP segmentation to the NIC, TSO (Section 2). The plot with legends '... and with NODELAY' refer to turning off Nagle's algorithm [71], which causes small messages to be aggregated by TCP. Nagle's algorithm makes more efficient use of the available bandwidth but may result in some delay for small messages. From Fig. 11, all varieties of TCP have greater latency for message sizes below 64 KB. The lower latency of the two TCP varieties without hardware offloading, may be attributable to applying TSO to small message sizes, as the PCI overhead of the segmentation template is relatively large. Another factor in hardware offloading is that, because the CPU processor runs faster than that on the Intel NIC, as the processor is repeatedly called for CRC calculation for small messages, their latency will be reduced. Of course, there is a trade-off against available processing for the application. After 64 KB, those varieties of TCP with hardware offloading have lower latency compared to TIPC, because PCI overhead is relatively reduced. Unfortunately, in the TIPC implementation under test, the maximum message payload size was set in the TIPC header and at various points in the source code of the kernel module to 66000 B (though this is expected to change in later implementations). Therefore, for message sizes above this limit, it was necessary to make two system calls, whereas the TCP socket only involves one system call.

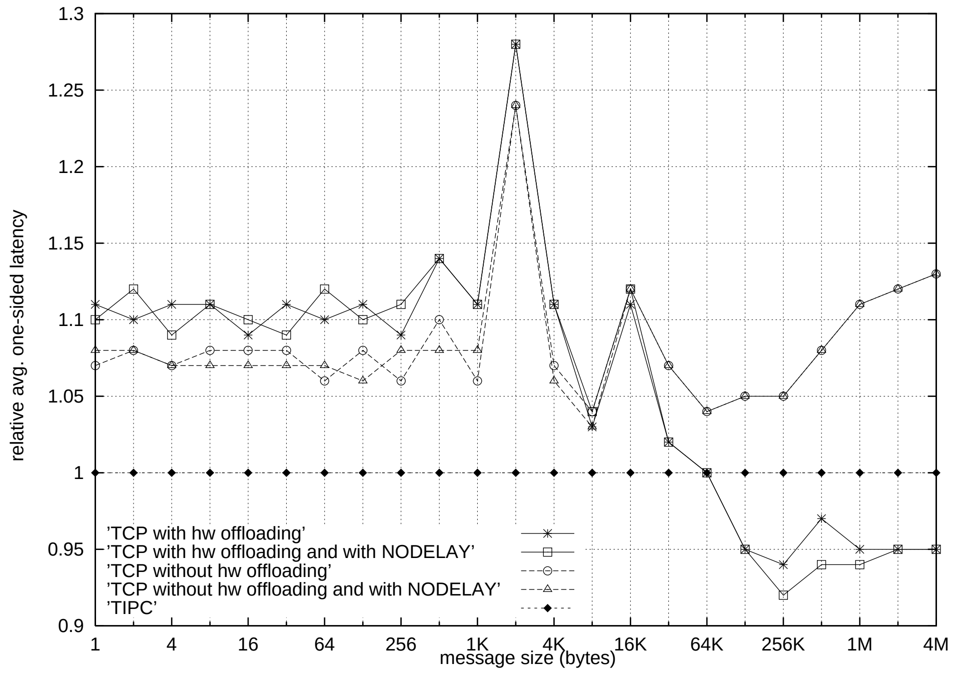


Figure 11: Comparative latency of TCP variants and TIPC across a range of message sizes, normalized to TIPC. (Note the log. scale on the horizontal axis.)

The payload throughput was measured for TIPC and TCP by finding the amount of data received within a tight receive loop over a pre-set interval. This somewhat underestimates TCP's total throughput, as the TCP/IP header takes up 40 B, whereas TIPC's header is 24 B, which size was confirmed by inspection with the Ethereal network protocol analyzer [72]. The throughputs are compared in Fig. 12, when it is apparent that the two varieties of TCP with hardware offloading outperform TIPC in that respect for larger message sizes. This can be ascribed to the gain from large-message hardware TSO. For smaller messages, Nagle's algorithm. when turned on allows TCP to register higher throughput. Message bundling does not appear to benefit TIPC to the same effect. Fig. 13, is significant as it shows that TIPC's system time (captured with the Unix utility `rusage` is dramatically better, for reasons developed in the introduction to this Section.

Turning to TIPC and TCP running under OMPI, Fig. 14 shows that for message-sizes below about 16 KB, TIPC has a clear advantage. There is also a sharp increase in TIPC latency at around 128 KB. TCP was able to employ an appropriate OMPI internal protocol, depending on message size. The same message parameterizations were retained in the TIPC BTL as in the TCP BTL. As previously mentioned, in the TIPC implementation under test, the maximum message payload size was internally set to 66000 B. To avoid attempts by

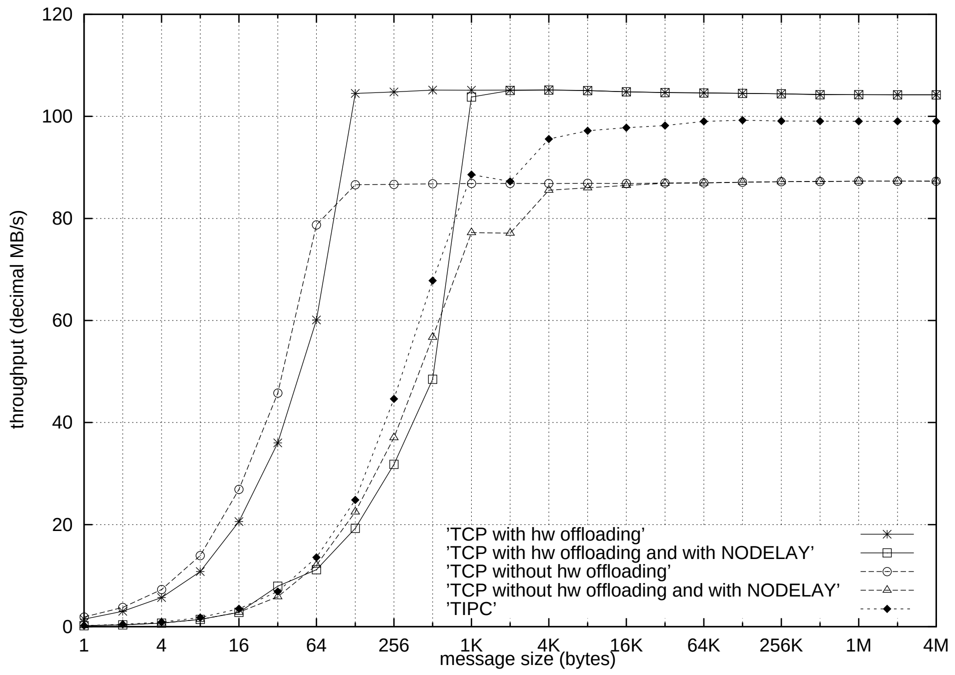


Figure 12: Throughput of TCP variants and TIPC across a range of message sizes. (Note the log. scale on the horizontal axis.)

OMPI to schedule larger sends in the pipelined version of RDMA, the third internal protocol for larger messages was disabled in the TIPC BTL, and the send protocol maximum was set to 66000 B. For message sizes of 128 KB and above, TIPC performs up to three rendezvous operations (for the first eager fragment, a maximum send-sized portion, and any remainder via the send internal protocol), whereas the TCP BTL negotiates a single RDMA operation for the entire message. Within the latency range that TIPC has the advantage, there is a 'sweet-spot' at about 512 KB, possibly due to internal buffering arrangements.

The comparison was now taken at the application level. The NAS Parallel Benchmarks (version 3.2) [10] are a set of eight application kernels/programs, *viz.* CG, IS, LU, FT, MG, BT, and SP, and EP, with MPI source code. OMPI was compiled with Fortran 77 bindings to match the NAS source code (though it was found that FT would not compile with the g77 (version 3.4.6) compiler and, hence, is omitted).

In [6], it is stated that TIPC spends 35% less CPU time per message than TCP, up to sizes of 1 KB, when the host machine is under 100% load. This is an important practical advantage, as clearly if TIPC spends relatively less time processing that time is available for computation. The test is described more comprehensively in [73] p. 25. For message sizes up to 16384 B, there is little to choose between TCP and TIPC, whereas for larger messages still TCP under this measure was superior. Accurate recording of CPU load is available through the utility `cyclesoak` [74], which can also act as a source of load.

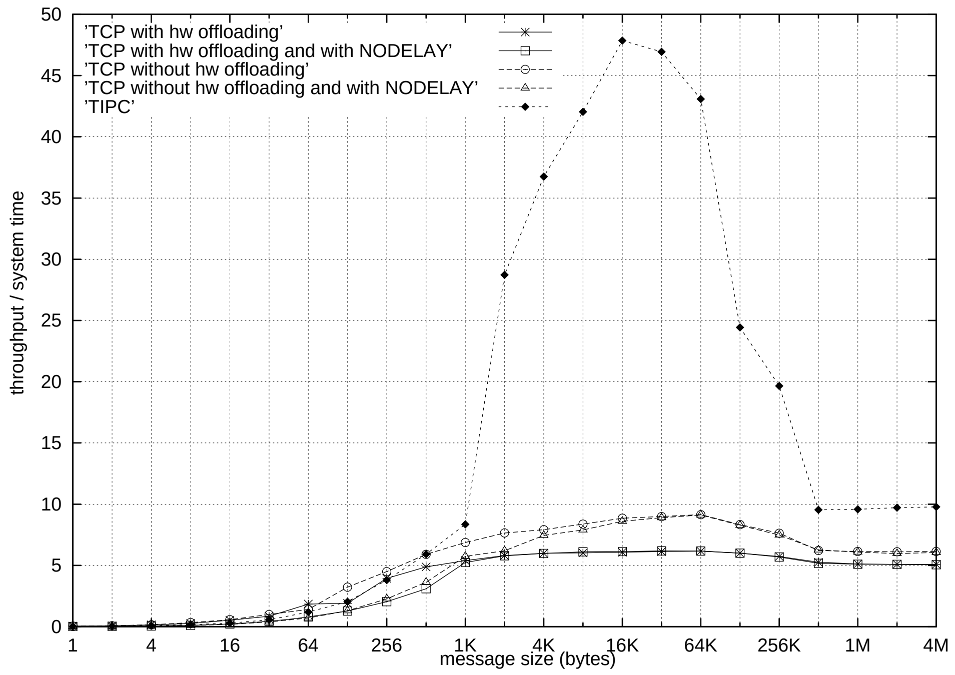


Figure 13: Comparative resource usage of TCP variants and TIPC across a range of message sizes, normalized to TIPC. (Note the log. scale on the horizontal axis.)

Following from this finding, two sets of tests were conducted, with and without load. In the tests recorded in Table 4, the background load was a `cyclesoak` process. In fact, the EP benchmark could also be adapted as a 'timewaster', as it continually creates pseudo-random numbers. Notice that as EP is CPU-bound, it only communicates at the end of its operations.

The W class (a measure of the dimension of the problem) test results appear in Table 4. In these tests, the best of five runs was taken rather than the median of five runs. This was because one of the application kernels, the IS integer sort, was found to be particularly sensitive to any system activity. For the others, the median does not differ noticeably from the best. LU and SP are computational fluid dynamics, while the others are diverse 'kernels'. For example, MG is a multigrid kernel for solving a 3D Poisson Partial Differential Equation set, being a hierarchical version with more rapid convergence than for the algorithm tested in Section 3.7. The Table also includes results for the MPICH version of MPI, with `ch_p4` being the standard MPICH method of spawning remote processes via the Unix utility `rsh`. The first two rows of Table 4 are taken from Table 2 of [75], also for 16 processors and W class problems. It will be seen that the message sizes are large for some of the tests, though for LU and MG they are about 1 KB and below the MTU.

From the results, it will be seen that MPICH almost always under-performs OMPI. Noticeable gains in performance occur for the TIPC variant of OMPI in the LU and MG

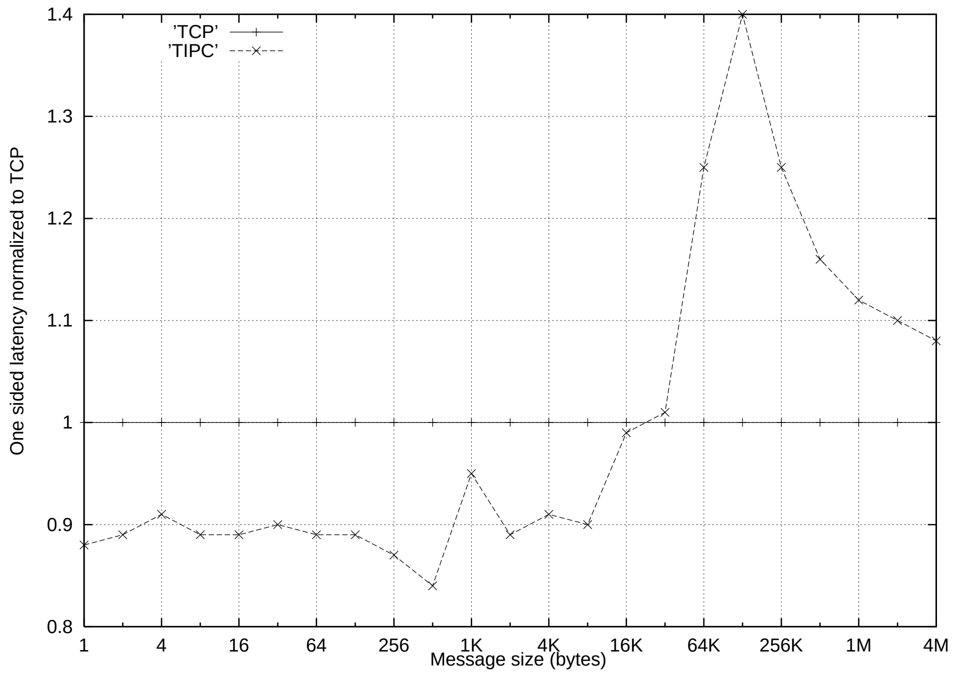


Figure 14: Comparative latency of OMPI/TCP and OMPI/TIPC across a range of message sizes, normalized to TCP. (Note the log. scale on the horizontal axis.)

application, both with shorter messages. Lastly, TIPC's relative performance compared to TCP increases when there is background load, suggesting an efficient stack.

As a comparison with the results for the application benchmarked in Section 3.7, the speed-ups are recorded for up to 23 physical nodes (the other nodes were unavailable at the time of the test). Clearly in Fig. 15, the effect of application loading balance is a significant increase in performance. Less easy to distinguish is the relative performance of OMPI/TCP and OMPI/TIPC, which illustrates the application-type dependency evident in the results for the NAS benchmarks, in this Section and in Section 6.

6 Data-link layer software

Interest in ULNs, allowing more direct application intervention communication processing has occurred over a period of time for a number of reasons, with examples being [68][76][77]. It was realized that traditional stacks were simply too cumbersome for parallel computing. The advent of fast and Gb Ethernet on clusters raised similar questions. In shared-memory multiprocessors, a single network interface was a bottleneck, and more recently multi-cores have also led to the suggestion of network channels with zero-copy payload transfer [78].

	Application benchmark					
	CG	IS	LU	MG	SP	EP
Comms. freq. (MB/s)	36.24	22.10	5.58	20.21	19.50	0.00
Comms. freq. (# msg./s)	6103	2794	5672	13653	1657	16
Avg. msg. size (B)	5938	7909	983	1480	11768	
mpich/ch_p4	436.65	88.18	4134.11	2140.89	1698.00	2140.89
OMPI/tcp	493.16	91.49	4688.89	2220.64	1792.43	141.84
OMPI/tipc	503.36	90.77	4921.43	2280.77	1812.18	140.88
tipc rel. tcp (%)	+2.07	-0.79	+4.96	+2.71	+1.10	-0.68
ch_p4 rel. tcp (%)	-11.46	-3.62	-11.83	-2.71	-3.59	-0.69
mpich/ch_p4 (loaded)	474.26	90.44	4137.57	2152	1686.01	141.55
OMPI/tcp (loaded)	494.70	91.56	4021.40	2237.23	1522.23	141.62
OMPI/tipc (loaded)	506.31	91.54	4356.74	2308.87	1574.66	141.71
ch_p4 rel. tcp (%)	-4.13	-1.22	+2.89	-3.77	+10.76	-0.05
tipc rel. tcp (%)	+2.35	-0.02	+8.34	+3.20	+3.44	+0.06

Table 4: NAS W class benchmark results (MOP/s) for sixteen processors, including relative (rel.) performance.

ULNs can be divided into those that take advantage of programmable Network Interface Cards (NICs) [79] and those that do not [14]. Some ULNs such as the Virtual Interface Architecture (VIA) [80] provide both varieties, with M-VIA being specialized to Linux. Though the ability to avoid temporary copies by processing the packet header at the NIC is an advantage, that advantage is tempered by the relatively slow speed of the embedded NIC processor, especially during periods of link congestion, or the need to re-program the NIC firmware [81]. Due to the exigencies of the mass market, the number of Gb Ethernet programmable NICs (NetGear GA620 and 3COM 3c985) is on the decline.

In fact, kernel level modules/agents have their own advantages because process scheduler overhead is reduced and the packet handling code can be written in a monolithic fashion. Even in ULNs there is a need to inform the kernel of data arrival for process safety reasons. It has also been found [14] that other factors apart from direct access to the NIC are important such as type of PCI bus, NIC driver, and the maximum transmission unit (MTU). An advantage of programmable NICs is the ability to transfer the packet payload directly to user space, rather than first place the payload in kernel buffers. In [15], zero-copy transfer was accomplished by ‘page flipping’, complex manipulation of the pages table. In [81], the gain from such manoeuvres is said to be overestimated, as measurements do not account for the gain from placing the data in cache, resulting from the first transfer.

Typically, when an Ethernet controller successfully receives a packet it generates an interrupt. The CPU then activates a device driver, which enqueues the packet on a backlog queue for later processing by an appropriate kernel protocol thread. Unfortunately, in times of severe link congestion, this can lead to the CPU spending all its time servicing interrupts and no time processing the enqueued packets. The problem still occurs even if a driver

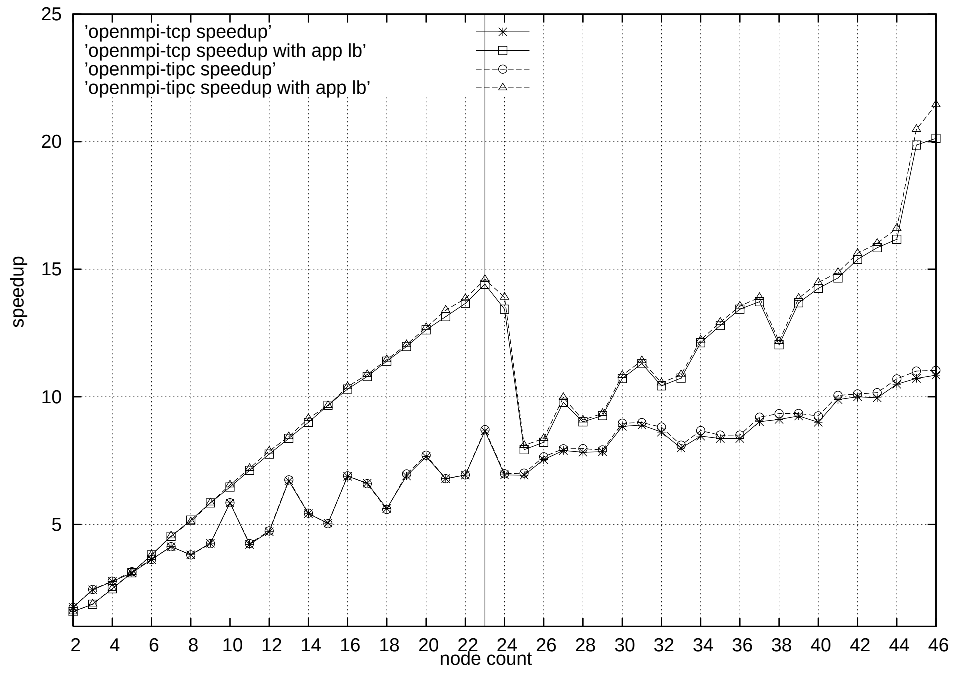


Figure 15: Comparative speedup of OMPI/TCP and OMPI/TIPC, with and without application load balancing.

batches interrupts by removing multiple packets, if they are available, whenever an interrupt occurs. This condition is known as receive deadlock [82], and is especially acute for Gb Ethernet interfaces with a potential for a rate of 80000 packet/s. On the other hand, if interrupts are delayed in periods of moderate link traffic, packet latency increases. To avoid this problem in Linux, the New API (NAPI) [83] can be set to reduce the rate of interrupts, by substituting clocked interrupts or polling, in a scheme originating from [82]. A by-product is that arriving packets remain in a DMA send ring (described in [15]), awaiting transfer over the PCI bus.

Thus, in the tests already described, NAPI `poll rx` was activated in the Linux e1000 driver, allowing a protocol processing thread to directly poll the DMA ring for packets. In fact, by judging link congestion, NAPI is able to maintain a flat packet service rate once the maximum loss-free receive rate has been reached, reactivating NIC-generated interrupts if need be. Performance tests [83] show a 25-30% improvement in per packet latency on a Fast Ethernet card when polling is turned on under NAPI. The 82540EM provides absolute timers which delay notification for a set period of time. The 82540EM also provides packet send/receive timers which interrupt when a link has been idle for a suitably long time [13]. Interrupt throttling is available, allowing interrupts to be suppressed if the interrupt rate goes above a maximum threshold.

The Genoa Active Message MAchine (GAMMA) communication layer [14] employs

NIC-dependent operations through a kernel-level agent. These operations include enabling / disabling interrupts, reading the NIC status, and polling the NIC. The code is highly optimized: with inline macros rather than C functions, and receiving messages in a single and immediate operation, without invoking the scheduler. GAMMA has a 'go-back N' internal flow control protocol. GAMMA has been ported to MPI, albeit with a customized version of MPICH protocols. GAMMA also utilizes the same method as `ch_p4` for remote spawning of MPI processes. However, GAMMA does require a second Ethernet NIC to be present for spawning purposes. Due to available resources, this restricted the tests in Table 5 to two cluster nodes. In these tests, the two nodes were connected by a crossover cable, and, hence, there is no switch latency. The results for NAS benchmark SP are not reported, as SP requires a perfect square as the number of nodes. In these tests, the cyclesoak utility [74] provided background load for the results marked 'loaded' in Table 5.

From the Table, it is apparent that GAMMA outperforms OMPI with both TCP and TIPC, when there is no background load. These results should be weighed against the NIC-dependent nature of the GAMMA implementation. TCP's performance is down in all cases and is relatively weaker than the sixteen processor results in Table 4. A feature of the results under loading is that OMPI with TIPC competes with GAMMA and out-performs OMPI with TCP. However, the same could be said about the MPICH results, implying that the problem is weaker performance by GAMMA under load.

	Application benchmark				
	CG	IS	LU	MG	EP
mpich/ch_p4	270.20	25.52	1029.89	552.22	18.17
mpich/GAMMA	307.70	28.29	1148.48	598.45	18.23
OMPI/tcp	213.25	15.84	922.54	486.32	18.33
OMPI/tipc	265.30	28.08	1055.94	551.91	18.17
mpich/ch_4 rel. tcp (%)	+26.17	+61.11	+11.64	+13.55	-0.16
GAMMA rel. tcp (%)	+44.29	+78.60	+24.49	23.06	+0.71
tipc rel. tcp (%)	+24.41	+77.27	+14.46	13.49	-0.16
mpich/ch_p4 (loaded)	269.26	25.63	1029.72	551.20	17.35
mpich/GAMMA (loaded)	277.32	23.27	965.98	549.79	17.32
OMPI/tcp (loaded)	195.51	15.80	868.96	458.10	17.32
OMPI/tipc (loaded)	243.74	27.13	990.47	522.73	17.31
mpich/ch_4 rel. tcp (%)	+37.72	+62.22	+18.50	+20.32	+0.17
GAMMA rel. tcp (loaded) (%)	+41.84	+47.28	+11.17	+20.02	0.00
tipc rel. tcp (loaded) (%)	+24.66	+71.71	+13.98	+14.11	-0.06

Table 5: NAS W class benchmark results (MOP/s) for two processors, including relative (rel.) performance.

7 Conclusion

Exploiting medium-scale Linux clusters for scientific problems is probably easily accomplished but to do so well is another issue altogether. A Linux cluster compared to a grid submission gives the advantages of ownership and accessibility. However, whereas it is obvious that off-the-shelf general purpose processors have made exponential gains in performance, due to Moore's law and the economies of scale in a mass market, it is almost axiomatic that until recently this was not the case in the communication inter-connect. The advent of Gb Ethernet has remedied that situation, though compared to custom high-speed interconnects, it is handicapped by the need for Ethernet framing. It is well known that communication software may contribute to a failure to achieve the latency and bandwidth capacity promised by the interconnect, which has been called the ”throughput preservation problem” [84].

The experiences reported in this paper suggest a number of conclusions. MPI, which is highly portable across a variety of machines, also presents competitive performance compared to Charm++ and MOSIX, two load-balanced alternatives. In the version of MOSIX under test, this performance was disappointing, a conclusion that extends to the communication primitives that underlie MOSIX's single system image programming model. It should be stressed that this does not allow firm conclusions to be made, as MOSIX versions are evolving. The application tests also found that if there is a preparedness to include application-level load-balancing (in combination with existing system-level load balancing), then speed-up can notably improve upon a plain MPI implementation. This becomes apparent once parallel slackness (more parallel processes than physical cluster nodes) is applied. The tests comparing TCP and TIPC revealed that TIPC has very real advantages over TCP, both within and outside an OMPI environment. This was the paper's strongest conclusion, as it was shown, for low-level benchmarks and for NAS application kernels, that short message latency was reduced. Moreover, the efficiency of the TIPC stack was demonstrated for nodes with high background load. The TIPC protocol has only recently been transferred to the Linux kernel and it is clear that further improvements are possible, such as its single message size limit. In high throughput applications with large messages, TCP is still to be preferred, as hardware offloading of CRC and TSO on the Gb Ethernet NIC speed up overall protocol processing. This advantage is unlikely, as yet, to be transferred to a protocol with far more limited deployment than TCP. The GAMMA user-level network interface (with MPI) is also capable of much improved performance over a combination of TCP and MPI. However, on a heavily-loaded machine, its performance may be matched by OMPI and TIPC. This implies that OMPI and TIPC are a more efficient solution from an application's point of view.

Acknowledgment

Dr David J. Johnston is thanked for earlier benchmarking of the Linux cluster.

References

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

[2] J. D. Day. The (un)revised OSI reference model. *Computer Communications Review*, 25:39–55, 1995.

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message-Passing Interface*. MIT, Cambridge, MA, 1994.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[5] M. Bhandarkar, L. V/ Kalé, E. de Sturler, and J. Hoeflinger. Object-based adaptive load balancing for MPI programs. In *International Conference on Computational Science*, pages 108–117, 2001. LNCS # 2074.

[6] J. P. Maloy. TIPC: Providing communication for Linux clusters. In *Linux Symposium*, volume 2, pages 347–356, 2004.

[7] 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.

[8] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, pages 91–108, 1993.

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

[10] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, NASA Ames Research Center, Moffet Field, CA, 1995. Report NAS-95-020.

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

[12] 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.

[13] Interrupt moderation using Intel Gigabit Ethernet controllers. Technical report, Intel Corporation, 2003. Application note AP-450.

[14] G. Ciaccio, M. Ehlert, and B. Schnor. Exploiting Gigabit Ethernet for cluster applications. In *27$\underline{th}$ IEEE Conference on Local Computer Networks, LCN'02*, pages 669–678, 2002.

[15] A. Gallatin, J. Chase, and K. Yochum. Trapeze/IP: TCP/IP at near gigabit speeds. In *USENIX'99 Technical Conference*, 1999.

[16] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

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

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

[19] 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), 1998.

[20] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Parallel and Distributed Systems*, 11(7):760–768, 2000.

[21] A. Barak and A. Braverman. Memory ushering in a scalable computing cluster. *Microprocessors and Microsystems*, 22(3-4):175–182, 1998.

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

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

[24] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, Cambridge, MA, 1996.

[25] B. Ramkumar and L. V. Kalé. Machine independent AND and OR parallel execution of logic programs: Part I-The binding environment. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):170–180, 1994.

[26] B. Ramkumar and L. V. Kalé. Machine independent AND and OR parallel execution of logic programs: Part II-Compiled execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):181–192, 1994.

[27] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems"*. MIT, Cambridge, MA, 1986.

[28] Parallel Programming Laboratory, University of Illinois, Urbana. *The Charm++ Programming Manual Version 5.4*, 2001.

[29] 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.

[30] W. Gropp, S. Huss-Lederman, A. Lumsdains, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT, Cambridge, MA, 2nd edition, 1998.

[31] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Supercomputing Symposium*, pages 379–386, 1994.

[32] J. M. Squyres and A. Lumsdaine. A component architecture for LAM/MPI. In *10$\underline{th}$ PVM/MPI Users' Group Meeting*, pages 379–387, 2003. LNCS # 2840.

[33] Open Systems Laboratory, Indiana University. *The LAM/MPI User Guide v. 7.1.1*, 2004.

[34] J. M. Bjørndalen, O. J. Anshus, B. Vinter, and T. Larsen. Configurable collective communication in LAM-MPI. In J. Pascoe, P. Welch, R. Loader, and V. Sunderam, editors, *Communicating Process Architectures 2002*. IOS Press, 2002.

[35] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[36] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22:1513–1526, 1997.

[37] N. J. Nevin. The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputing Center, Columbus, Ohio, 1996.

[38] N. Nupairoj and L. Ni. Performance evaluation of some MPI implementations on workstation clusters. Technical report, Dept. of Computer Science, Michigan State University, 1996.

[39] S. B. Kim S. Markus, K. Patazapoulos, E. N. Houstis A. L. Ocken, P. Wu, S. Weerawarana, and D. Maharry. Performance evaluation of MPI implementations and MPI based parallel ELLPACK solvers. In *2$\underline{nd}$ MPI Developers Coneference*, pages 162–168, 1996.

[40] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *16$\underline{th}$ International Workshop on Languages and Compilers for Parallel Computing*, 2003. Paper # 03-07.

[41] C. E. Korman and I. D. Mayergoyz. A globally covergent algorithm for the solution of the steady-state semiconductor device equations. *Journal of Applied Physics*, 68(3):1324–1334, 1990.

[42] H. K. Gummel. A self consistent iterative scheme for one-dimensional steady-state transistor calculations. *IEEE Transactions on Electron Devices*, 11:455–456, 1964.

[43] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, Int. Conf. on Computational Science (ICCS)*, 2003.

[44] Z. Balaton, P. Kacsuk, and N. Podhorszki. Application monitoring in the grid with GRM and PROVE. In *International Conference on Computational Science*, pages 253–262, 2001.

[45] 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.

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

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

[48] W. R. Stevens. *UNIX Network Programming*, volume 2: Sockets and XTI. Prentice Hall, Upper Saddle River, NJ, $2^{\underline{nd}}$ edition, 1999.

[49] M. J. Rochkind. *Advanced Unix Programming*. Addison-Wesley, Boston, MA, $2^{\underline{nd}}$ edition, 2004.

[50] W. R. Stevens. *UNIX Network Programming*, volume 2: Interprocess Communication. Prentice Hall, Upper Saddle River, NJ, $2^{\underline{nd}}$ edition, 1999.

[51] J. Maloy, S. Blake, and M. Koning. TIPC: Transparent Inter Process Communication Protocol, 2004. IETF Internet Draft, expired July, 2004.

[52] R. Martin, A. Vahdat, D. Culler, and T. Andersen. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *$24^{\underline{th}}$ Annual International Symposium on Computer Architecture*, 1997.

[53] F. Chong, R. Barua, J. D. Kubiatowicz, and A. Agarwal. The sensitivity of communication mechanisms to bandwidth and latency. In *$4^{\underline{th}}$ International Sysmposium on High Performance Computer Architecture (HPCA-4)*, pages 37–46, 1998.

[54] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the Virtual Interface Architecture. In *Supercomputing Conference*, pages 1–15, 1998.

[55] M. Abbot and L. Petersen. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, 1993.

[56] T. Rühl, H. Bal, R. B. Hoedjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDTPA'96)*, pages 1477–1488, 1996.

[57] M. Koning. An introduction to TIPC. In *Multicore Expo, CA*, 2006. Slides available at TIPC's SourceForge page.

[58] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 2960: Stream Control Transmission Protocol, 2000.

[59] C. Dubnicki, L. Iftode, E. W. Felton, and K. Li. Software support for virtual memory-mappped communication. In *International Conference on Parallel Processing Symposium*, pages 377–381, 1996.

[60] J. M. Squyres and A. Lumsdaine. A component architecture for LAM/MPI. In *10$\underline{th}$ European PVM/MPI Users' Group Meeting*, 2003. LNCS No. 2840.

[61] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovski, and J. J. Dongarra. Fault tolerant communication library and applications for high performance. In *Los Alamos Computer Science Institute Symposium*, 2003.

[62] D. E. Bernholdt et al. A component architecture for high-performance scientific computing. *International Journal of High-Performance Computing Applications*, 20(2):163–202, 2004.

[63] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4):285–303, 2003.

[64] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High performance custering technology. *IEEE Micro*, 22(1):46–57, 2002.

[65] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G .E. Fagg. The open run-time environment (openrte): A transparent multi-cluster environment for high-performance computing. In *12$\underline{th}$ European PVM/MPI Users' Group Meeting*, 2005.

[66] B. W. Barrett, J. M. Squyres, and A. Lumsdaine. Implementation of Open MPI on Red Storm. Technical Report LA-UR-05-8307, Los Alamos National Laboratory, Los Alamos, New Mexico, USA, 2005.

[67] W. Yu, T. S. Woodall, D. K. Panda, and R. L. Graham. Design and implementation of Open MPI over QsNet/Elan4. Technical report, Ohio State University, OH, 2005. Report no. OSU-CISRC-10/04-TR54.

[68] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *19th Annual International Symposium on Computer Architeture*, pages 256–266, 1992.

[69] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control for fast, long distance networks. In *IEEE INFOCOM*, pages 2514–2524, 2004.

[70] R. L. Cotterell, H. Bullot, and R. Hughes-Jones. Evaluation of advanced TCP stacks on fast long-distance production networks. In *Workshop of Protocols for Fast Long Distance networks*, 2004.

[71] J. Nagle. Congestion control in IP/TCP interworks, 1984. RFC 896.

[72] G. Ramirez, B. Caswell, and N. Rathaus. *Nessus, Snort, & Ethereal Power Tools*. Syngress, 2005.

[73] J. Maloy. Telecom Inter Process Communication. Technical report, Ericsson Ltd, 2003. Report No. LMC/0–01:006.

[74] A. Morton. Cyclesoak — a tool for measuring resource utilization, 2003. Available from `http://www.zipworld.com.au/ akpm/linux/`.

[75] K. Morimoto, T. Matsumoto, and K. Hiraki. Performance evaluation of MPI/MBCF with the NAS Parallel Benchmarks. In *6th European PVM/MPI Users' Group Meeting*, pages 19–26, 1999.

[76] C. Csanady and P. Wyckoff. Bobnet: High-performance message passing for commodity networking components. In *International Conference on Parallel and Distributed Computing and Networks*, 1998.

[77] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit Ethernet message passsing. In *Supercomputing Conference*, 2001. Paper 57.

[78] V. Jacobson and R. Felderman. Speeding up networking. In *Linux Conference Australia*, 2006. Talk given at the conference.

[79] P. Balaji, P. Shivan, P. Wyckoff, and D. Panda. High performance user level sockets over Gigabit Ethernet. In *IEEE International Conference on Cluster Computing, Cluster'02*, 2002.

[80] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–75, 1998.

[81] I. Pratt and K. Fraser. Arsenic: A user-accessible Gibabit Ethernet interface. In *IEEE INFOCOM*, pages 67–76, 1999.

[82] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[83] H. Salim J, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *5th Annual Linux Showcase & Conference*, pages 165–172, 2001.

[84] 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.