All About Trollius

Greg Burns
Vibha Radiya
Raja Daoud
Raghu Machiraju
The Ohio State University
Columbus, Ohio

An Opportunity for a Standard?

Trollius was conceived because the best multicomputer software in existence at the time of its birth, Autumn 1985, was not very appealing even to the adventurous computational scientist. In particular, the transputer, building block of a new generation of multicomputers, had essentially no operating system. Trollius started with both a general and a specific purpose.

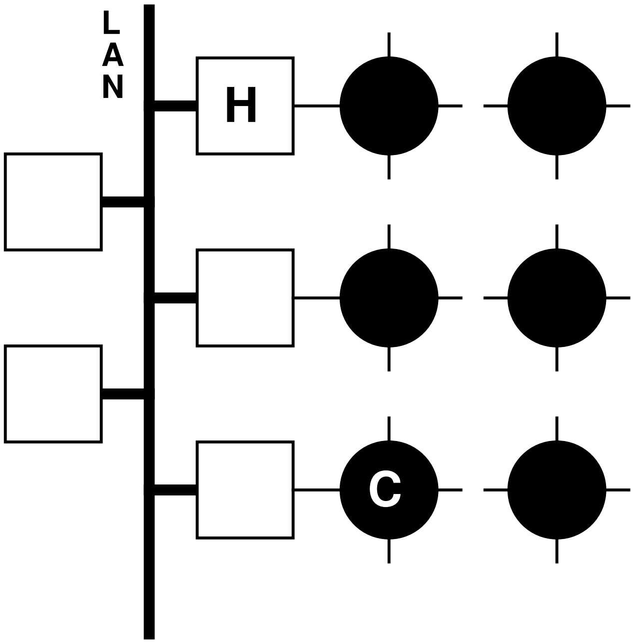
An opportunity appeared to build a strong operating system foundation suitable for future multicomputers. Contemporary system software from the hypercube generation appeared to be hastily constructed solutions from companies under pressure to reach the market fast and focus attention on the promise of the architecture. Since the first commercial introduction of multicomputers, hardware concerns have been progressively addressed while software has lagged behind. Standards have not appeared. Trollius did not anticipate a multicomputer as a stand-alone workstation, but virtually every other possible usage is supported. New operating systems continue to be built, often for proprietary reasons, and the lowest common denominator remains stalled at the message passing multicomputer architecture. Until higher platforms are accepted, progress will be slow.

The Trollius Project is structured for maximum impact on the young field of concurrent processing. Within the university environment, the development team at The Ohio State University has the latitude to put engineering first and ignore the bottom line. Away from academic departments, the developers are full-time professional engineers, concerned with building an industrial strength product first and tackling research issues second. Trollius is the only operating system portable to any multicomputer incarnation and available in source form. In particular, getting the source into the hands of bright students and faculty at other universities has been its formula for survival. People can use other systems and become expert users. People can perform surgery on Trollius and become guru developers. This article will point out that the structure of Trollius endeavors to make users and new doctors comfortable and uninhibited.

Mission for a Perceived Supercomputer

Trollius was born, raised and continues to live in the house of U.S. supercomputer centers. From this perspective, the ultimate purpose of scalable parallel architectures is to extend the limits of supercomputing and permit attacks on the grand challenges of computational science. Thus, the first concern of Trollius has been to support data parallel applications. Many other system and language solutions emphasize their elegance and efficiency at solving function parallel applications. Trollius has concentrated higher level development on data parallelism, but all the building blocks are in place for any strategy of multicomputer problem solving.

Is executing the designated application the only role an operating system must play? If the answer is yes, then a network loader and a communication library might be the final solution, because multicomputer applications are typically not very complex - today. Trollius anticipates that the answer is no. Monitoring, debugging, machine maintenance and machine portability must also be considered. If third party vendors and university research groups are essential



The General Multicomputer - Trollius is a uniform foundation for message passing on every multicomputer node. Host nodes (H) run Trollius with the local operating system while compute nodes (C) run it natively.

to software progress on new architectures, then a gutsy operating system must not set limits and must not attempt to predict the future but instead must concentrate on the accepted basics. The alternative is more new operating systems starting from scratch instead of the shoulders of others.

The basic activity of a multicomputer is passing messages between processes running on nodes. Other activity is an abstraction, a paradigm, a package. Abstractions are great creative things, but the basics need to be accessible to the programmer in a direct, no-nonsense, complete manner, like the standard TTL NAND gate is to the digital engineer, in case one of the higher level packages does not fit the need. No one abstraction is the answer to every problem solving strategy on the multicomputer architecture. A thorough package of the basics supporting any possible abstraction is the answer.

The Golden Rules

The previously discussed motivation and purpose spawned the principles of Trollius's design, listed succinctly below:

- Focus on supercomputer applications, scalable problems and Unix.
- Expand the multicomputer scope to include the local area network.
- Utilize an onion-like design, in which each layer can be peeled off for less functionality but better performance.
- Always allow the user to access the basics: nodes, processes, links and messages and finally, the hardware.
- Maintain node uniformity; host and compute nodes run the same Trollius.
- Engineer modular software that facilitates other development. Structure with multiple processes above a spartan message passing local node kernel.
- Don't bet on any one compiler, processor or topology.

Structure and Functionality

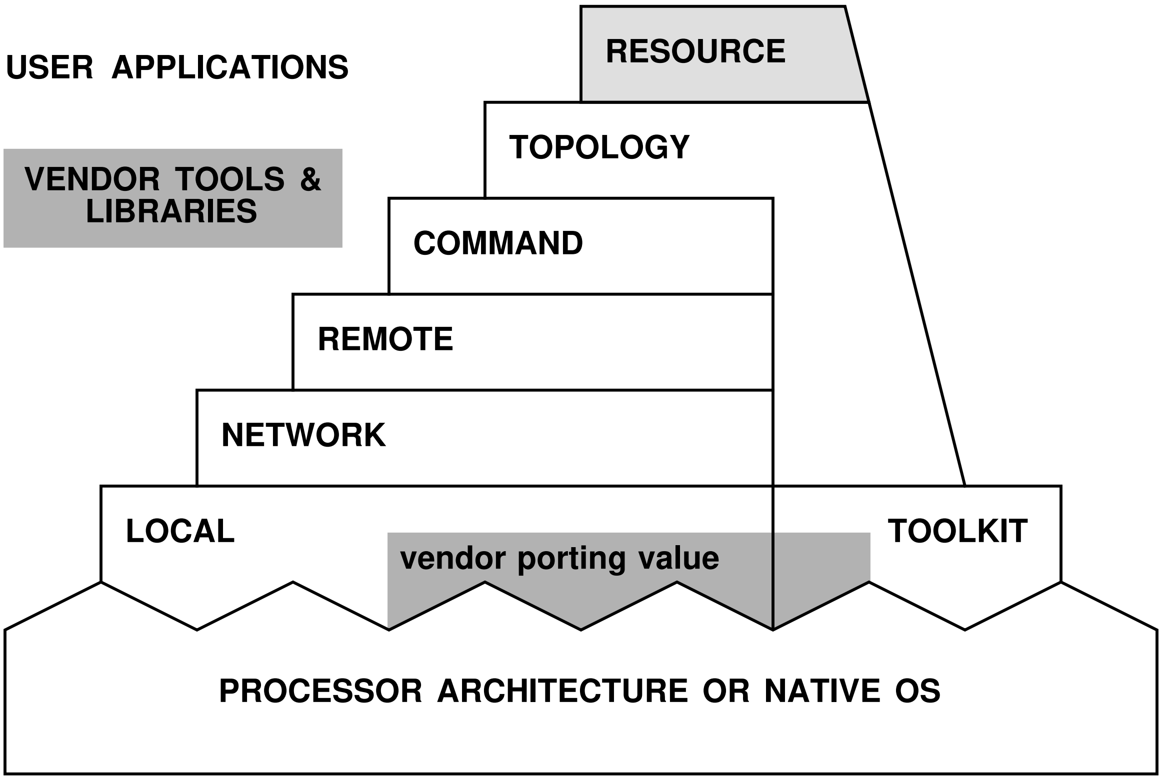
To dispel the suspicion that Trollius is a primitive operating system, the functional description that follows begins at the current top of the evolutionary tree and works downward.

Crystalline (CrOS) is a communication library for domain decomposition developed at the California Institute of Technology. It bypasses the modest, yet annoying problems of grid programming: simple deadlock and global singularities. CUBIX is a file access method, also developed at Cal Tech, that collects and distributes data among multiple nodes in an orderly manner. CUBIX often precludes the need to write a control program for the host or root node. Together, CrOS and CUBIX are the recommended tools for grid domain decomposition; Trollius offers both.

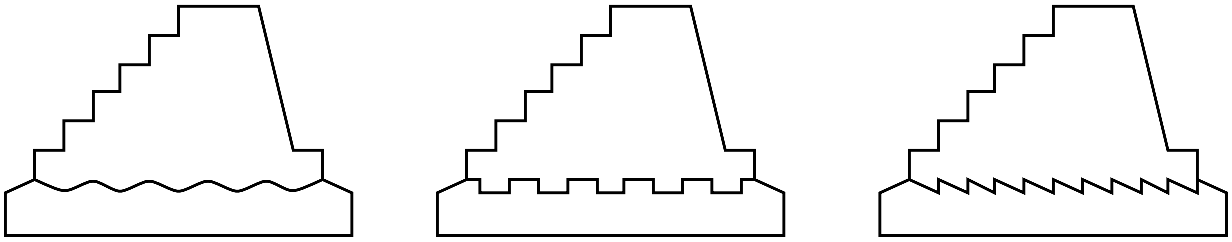
Topology Layer

Ready to boot the multicomputer, the user confronts the highest layer of Trollius 2, the topology layer. The topology layer tools interpret a simple grammar called Node and Interconnect Language (NaIL), in which an exact description of the multicomputer to be booted has been recorded. Given this topology description, called the boot schema, Trollius endeavours to completely solve the general problem of link configuration, booting and routing with very few limits. In particular, topology is not limited. The boot schema describes the number, type (compute node, host, disk connected, CRT connected, wasted interface) and identification (any 32 bit quantity) of multicomputer nodes. The one restriction, removed in Version 3, is that only one host node may exist. Also described are the links between nodes, explicit routes to be taken between two points, priority route assignment between two points, and the correspondence between Trollius node identifiers and vendor specific naming conventions (so that vendor specific link configuration tools may be invoked). Most common configurations for a particular installation will be created by the system administrator and saved.

Given multiple links between the host node and the compute node pool, Trollius 2 permits multiple user access to the pool. Trollius does not allow sharing of single links or single compute nodes. Multiple user access in Version 2 is managed by personal communication. This illustrates an important policy of Trollius development - each release selects a cut-off point and provides the complete implementation of the basics to that point. Trollius 1 had no topology layer and the cut-off point for complete solution was the local node operating system. The route table for each node was hard-wired. Booting was accomplished by invoking the individual node booting tools of the command layer (hboot (host node), sboot (device connected compute node) and vboot (remote compute node)) by hand. Today, the topology tools drive these command layer tools, and in Version 3, the resource layer tools will drive the topology tools. If a user ever needs to bypass the to-



Trollius Layered Structure - Each layer has a standard interface accessible to the user. Vendors can add proprietary value in porting to specific hardware and in bundling tools and libraries. The resource layer is part of the third generation of Trollius.



Portable Node Interface - The local and toolkit layers are portable to any compute node architecture. Other layers are unaffected by the underlying environment.

pology layer, the multicomputer can still be operated from the command layer.

Consider a 2-D problem in computational fluid dynamics solved with a finite difference method on a cartesian grid. Convinced of the merits of concurrent programming, the scientist parallelizes the code using CrOS and CUBIX. He/she has access to a 64 node transputer array connected to a Unix workstation. First, he must compile the program, as demonstrated below:

```
% tf77 -o cfd cfd.f -lcros
%
```

CUBIX is an integral part of the Trollius remote file system and is accessed through the standard library. CrOS is an optional library linked explicitly. This compilation can take place on any Unix machine with an installed and licensed tf77 cross compiler supported by Trollius.

The user selects or prepares a boot schema for a multicomputer configuration suitable for the CFD application. Trollius CrOS demands consecutive node identifiers beginning with 0 and continuing through the grid in row major order. Perhaps the boot schema for an 8x8 grid is called "bnail64.cros", a slight misnomer because there are 65 Trollius nodes, including the host node. The following three topology tools establish a running multicomputer:

```
% map -c bnail64.cros
% solder -c bnail64.cros
% spread -c bnail64.cros
%
```

The map step generates a file containing a route table for each node using a greedy algorithm based on accumulating link weights. The weights and more may be fine tuned or over-ridden in the boot schema. Once a route file exists for a given boot schema, the map step can be skipped. Map makes the general problem of routing completely detached from the Trollius Operating System. It is an off-line tool that can be run anywhere. With well defined input and output files, the creative inventions of individual researchers can replace map.

The solder step invokes the vendor specific software necessary to configure the communication links to match the topology described in the boot schema. Solder is not used with fixed links.

The spread step actually boots the multicomputer, invoking the right command layer booting tool for the right type of node and loading route and general network information into each node. Like all Trollius host programs, the user invokes spread from the Unix interface. Control returns to the same interface; no special interface or shell exists. After booting, the user's Unix workstation is also the origin host node in a Trollius multicomputer.

Command Layer

A host node program written with the Trollius communication library can cooperate with processes running anywhere in the multicomputer. Trollius supplies control and monitoring programs that make up the command layer. These programs supplement the extensive suite of familiar Unix development tools.

The user is ready to run the program, cfd, on the compute nodes using the loadgo command:

```
% loadgo -v C cfd
[1] 348013 cfd running on all compute nodes
%
```

Hands-on information gathering gives Trollius its flavour. For example, is the program running?

```
% state n0,63
NODE  INDEX  PID  KPRI  KSTATE   PROGRAM
n0    [1]    348  0     A (6, 0) cfd
n63   [1]    348  0     A (6, 0) cfd
%
```

Are the buffers getting clogged?

```
% bfstate -l n32
NODE  DEST  EVENT  TYPE  LENGTH  STATE
n32   n32   6      0     2048    blocked
n32   n32   6      0     2048    waiting

n32   max workers = 4, used workers = 2,
      full workers = 1
n32   max space = 262144, used space = 4096
%
```

How much memory is free? Perhaps the granularity of the data decomposition can be adjusted.

```
% mstate n7 -f
NODE  PID   ADDRESS   SIZE
n7    free  80000480  B80
n7    free  800535A8  1AD258
%
```

The application either terminates or the user may decide to deliver a termination signal with the doom command:

```
% doom C %1
% sweep C
%
```

Killing a message passing parallel application in mid-flight will often result in a number of unreceived messages in lingering transit. These messages will settle into buffers and can be inspected with the bfstate command shown above. Lingering messages can affect the next program run. The debris can be cleaned up with the sweep command. Then, the program can be run again without rebooting. Afterwards, the host node Trollius is terminated and the compute nodes released by the tkill command.

```
% tkill
%
```

The topology and command layers form the operational level of Trollius. The programming interface begins with the remote layer.

Remote Layer

Commands process options and present data surrounding one or more invocations to remote layer functions found in the remote library, trillium (Trollius's original name). Thus, any user program can accomplish the function of any command program. A remote function exchanges messages with a remote daemon. For example, the command bfstate calls the function rbfstate(), which talks with the daemon, bufferd, on the indicated node.

The daemon called upon most often by user programs is filed, which provides a remote interface to the Unix filesystem. Any compute or host node can and run filed if it operates a disk. The service includes a large number of standard Unix functions supporting the C stdio package, the Fortran Unit I/O system or direct access of portable Unix programs. (Note: This illustrates another important Trollius policy concerning Unix. Trollius is real Unix where it must be (the host nodes), Unix compliant where it makes sense (the filesystem), and not Unix where it would serve badly (communications)).

Filed only runs on nodes with attached disks. Other daemons only run on compute nodes, where Trollius is the native operating system and has more responsibility for basic services. The following table lists the other remote daemons, their responsibilities, and where they run.

DAEMON	NODE	FUNCTION
mallocd	compute	memory allocation, deallocation and monitoring
kenyad	compute	process creation, signalling, monitoring and spawning
memd	compute	fetch/load specific memory addresses
flatd	compute	load memory and assign tag to address
bootd	compute	boot neighbouring compute nodes
loadd	host	load data and program files on compute nodes
echod	all	echo messages

Network Layer

This excursion to the depths of Trollius has reached the layer at which CrOS is written, the network layer. If not the heart, the network layer is certainly the hub of the operating system. A group of processes that run on every node provide a network message passing service. At the network layer, Trollius functionality is completely uniform among all node types.

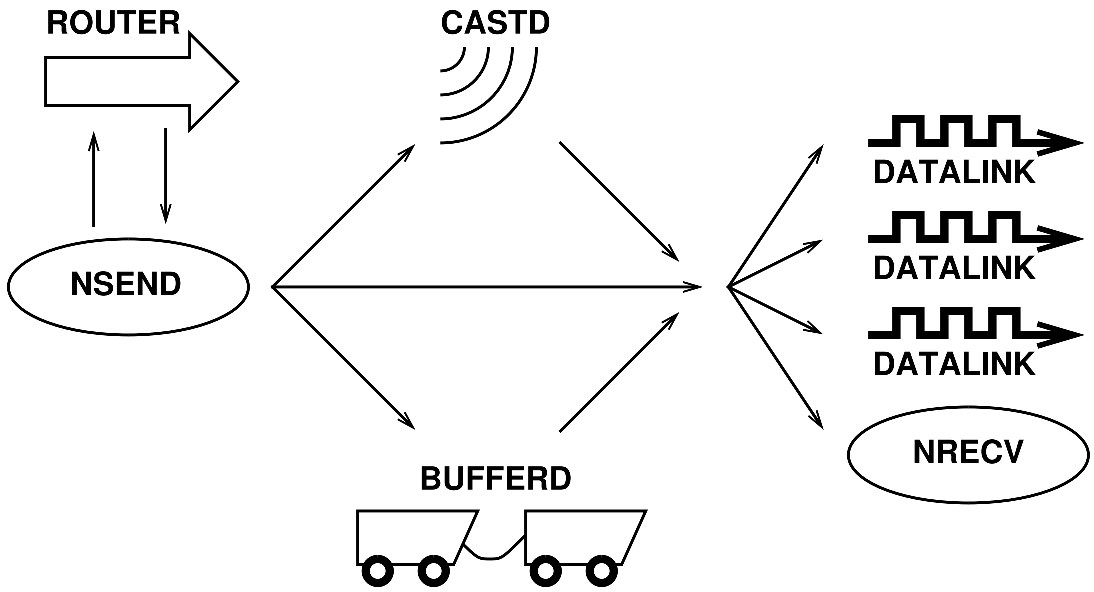
The smartest process is the router, which keeps the identifier, type and route to every other node in the system. One will be able to change this information at runtime in Version 3. A small cache enables local clients to avoid continuous, expensive communication with the router for each network message sent.

When a process wants to send a message to an arbitrary node, it invokes nsend(), which first consults its own route cache or the router. The obtained "route" indicates the local forwarding process, a process running on the same node that will receive the message and forward it towards its destination. Typically, the forwarding process is a datalink output process, the proprietor of an output link. Whenever a datalink output gets a message, it simply pumps the message out its link, which could be a transputer link, a Unix device driver or a TCP/IP connection. Datalink input processes wait for a Trollius network message to arrive on the same links. It calls nsend() on every message it receives and the procedure just described repeats itself. When a message has reached the correct node, the forwarding process is the receiving destination process specified by the original caller of nsend().

Once the forwarding process is known at each stage, nsend() actually moves the message to the forwarding process by calling dsend(). If a process knows the forwarding process in advance, it need not consult the router and can call dsend() directly.

At no point does any process consider dropping a network message. If dsend() detects that a forwarding process is busy, it must either wait or pick another forwarding process. If a datalink input must wait, no other message can be read from its link. A clogged link can be a bottleneck and can cut off control over a node by the user seated at the host node. A need exists for message buffers.

Bufferd is a network layer forwarding process that manages a pool of message buffers and a small group of worker processes. Unless forbidden by its caller, dsend() can forward its message to bufferd. The buffer daemon forwards the message to an idle buffer worker which then blocks until the original forwarding process is free. If no worker is available, the message waits in a large pool inside bufferd. When this is exhausted, callers to dsend() wait on bufferd. Users can tune the number of workers and the size of the message pool from the command line.



Network Forwarding Processes - After consulting the router, a process calling nsend() transfers the message to a local forwarding process. Bufferd can hold multiple messages, up to a space limit tuned by the user.

Another forwarding process is the cast daemon. Within the domain of physical node identifiers are cast identifiers which refer to groups of nodes. A few useful casts, like broadcast, are predefined, but all others are explicitly chosen and registered by user processes, which then use it as a normal destination node with nsend(). When nsend() consults the router about a cast identifier, it is directed to forward its message to castd. Castd then performs the necessary nsend() operations to the correct node neighbours which reflect an efficient spanning tree to the cast members. Castd is completely topology independent. Castd demonstrates the modularity of the Trollius network layer. Services can be unplugged, modified, and re-inserted with little concern for the rest of the system. Forwarding processes can be implemented for new network services and only the programmable router is the wiser.

Local Layer

The real messages in Trollius are the local or kernel layer messages. A network message comprises two kernel messages; the first contains the network message descriptor and the second contains the actual data. Forwarding a message with dsend() really means sending kernel messages to a local process with ksend(). Processes synchronize on two arbitrary numbers: a message event and a message type. Synchronizing events must equal in value and the types must match in at least one bit (bitwise AND is true) for a message to be transferred. The "indication of forwarding process" alluded to earlier is the correct event and type. Local message passing calls block a process until it synchronizes unless the associated ktry_send() polling calls are used.

Synchronization is managed by the kernel, which in Trollius is a server not an executive. The kernel is considered a server because it is not in complete control of the processor. Final scheduling and ultimate process creation/destruction is handled by the underlying environment: the host node's native operating system or the transputer's microcode. On other compute node processors, these responsibilities are handled by a software sub-kernel. Further explanation of the sub-kernel is left to another publication.

Although the rendez-vous between two processes occurs through the kernel, actual data is transferred directly, without the kernel's aid. The rendez-vous tells each process what it needs to know in the underlying environment to communicate directly with its peer (on the transputer, this would be an channel address). Directed by a flag, the ksend() runtime function can remember the results of synchronizing on multiple events and then bypass the kernel the next time a known event is indicated. The capability is called a virtual circuit which effectively welds a dynamic group of processes into a static group. When used between callers of nsend() and datalink processes, an efficient virtual circuit can exist between any two points in the multicomputer. The lower overhead of virtual circuits makes them the preferred communication mechanism for nearest neighbour, domain decomposed applications seeking finer granularity. (Note: CrOS has an option for virtual circuits). A circuit that crosses a link reserves that link for the caller of dsend(). This facilitates the lowest level call in the network layer, psend(), which drives the hardware directly.

The predominant responsibility of the spartan local kernel is message passing among dynamic local node processes. It does not know of multiple processors. Since many underlying environments do not supply a fully developed, predictable priority scheduling service, the Trollius kernel has an option to schedule processes (by controlling replies to kernel requests) to the underlying environment. When a client process attaches to the kernel it can decide to be uncontrolled, in which case scheduling is completely handled below. Alternatively, the client process can supply a priority, in which case Trollius guarantees that only the highest priority unblocked process runs.

Another kernel feature is asynchronous signal delivery modeled on Unix. Signals may only be caught during kernel requests because the kernel is a server. Clients can detach from the kernel and go about their business independently. The kernel's server design is the key to Trollius uniformity and portability between host and compute nodes.

Toolkit Layer

The toolkit layer is not formally part of the operating system because it does not deal in Trollius messages. Instead, the tools concentrate on booting the host nodes and device connected compute nodes. It would be possible to burn Trollius into PROMs on compute nodes, but again, a modular approach is followed. As with all other layers, the philosophy guiding the toolkit layer is to support other solutions that could fit into the next ascending layer, not just the one chosen by Trollius. Since the toolkit layer boots Trollius onto certain nodes, the same tools could presumably be used to boot anything.

A configuration file called a process schema indicates the programs that constitute the operating system on each type of node. The hboot tool uses one process schema to start processes on a host node. The sboot tool uses another processschema to start processes on a device connected compute node. Sboot() invokes the patch tool to relocate addresses (given a compute node with no dynamic address translation) and then the cboot tool to actually load a program. Cboot communicates with the Trollius bootstrap loader, called moses, running on the device connected compute node. Moses is loaded by the export tool, which talks to whatever boot mechanism exists after reset. Moses can also be burned into PROM. The reset is achieved with the fish tool, the simplest, lowest order of functionality in Trollius and a very long way from the spread tool.

How Fast Will It Go?

The following performance numbers were obtained on a T800 (20 MHz) using the LSTT C compiler under the latest version of Trollius (2.0 He). Both code and data resided in external memory. The measurements made between two nodes involved directly connected neighbours (and no C004 or other link reconfiguration switch).

The time per message per hop T is:

$$T = Ts + (Tc * L) + (Tp * Ep)$$

where

T	is	time per message per hop (usec.),
Ts	is	set up charge (usec.),
Tc	is	communication time (usec./byte),
Tp	is	packetization cost (usec./extra-packet),
L	is	message length (bytes), and
Ep	is	number of extra packets (after the 1st one)

and

$$Ep = \begin{cases} ceil(L/Lp) - 1 & \text{for } L > 0 \\ 0 & \text{for } L = 0 \end{cases}$$

where

ceil()	is	the integer ceiling function
Lp	is	the packet size (bytes) (4096).

Network messages are limited in size. Larger messages are packetized. The default packet size is 4096 bytes although this can be changed before compiling Trollius.

A high overhead is incurred when a message passes through a buffer. Buffers can prevent deadlock and reduce blocking. In practice, they tend to enhance the robustness of the system. Trollius does not choke if a process accidently sends 1000 messages to a node with no receiving process, but at a substantial cost. Message buffering can be disabled by setting the

NOBUF flag in the nh_flags field of the message header. The cost per message buffered is

Tb = 2375 + (0.20 * L) (in usec.).

At each communication layer, regular and virtual circuit message passing have been measured between two directed connected nodes (n0-n1) and within one node (n0). To help in comparing and deciding which of the communication layers is most suited for a particular application, a quick summary of the functionality provided by each layer is given below.

Network Layer (nsend)

- routed: from/to any node
  multitask: from/to any node process
  packetized: any message size

Network Datalink Layer (dsend)

- not-routed: nearest neighbour only
  multitask: from/to any node process
  not-packetized: message size limited to Lp (default: 4096)

Local Layer (ksend)

- not-routed: local communication only
  multitask: from/to any process on the same node
  not-packetized: no limits on message size but with protection against mismatched lengths

Network Physical Layer (psend)

- not-routed: nearest neighbour only
  not-multitask: single process communication
  not-packetized: no limits on message size but no protection against mismatched lengths
  restricted use: can only be used after a virtual circuit has been set up

REGULAR			VIRTUAL CIRCUIT			
Ts	Tc	Tp	Ts	Tc	Tp	
480	.56	552	168	.56	140	nsend n0-n1
388	.10	374	211	.10	195	nsend n0
453	.56	---	128	.56	---	dsend n0-n1
355	.10	---	177	.10	---	dsend n0
230	.10	---	69	.10	---	ksend n0
---	---	---	36	.56	---	psend n0-n1

During the development and debugging of an application program, use of nsend() is recommended. When the program is working, nsend() can be substituted for dsend(), ksend() or psend() where possible. Although this provides enough flexibility for most users, it is possible that none of the above suits a particular application. For such requirements, the project has developed the Custom Network Gearbox approach to tailor the communication functionality/performance to applications that fall into the cracks of the regular Trollius communication layers. This approach is explained in a paper published in the proceedings of the Fifth Distributed Memory Computing Conference.

State of the Project

The first version of Trollius was written at the Cornell Theory Center. The recent second generation Trollius (described here) and the upcoming third generation is based at The Ohio State University, Research Computing. The project also continues at Cornell with Dave Fielding, Moshe Braner, Jim Beers and Roslyn Leibensperger. Source level releases for academic sites are available from OSU for $300 US. Industrial sites are encouraged to make a larger donation. Two compiler options, the Logical Systems Transputer Toolset or Genesys are licensed separately and bundled with Trollius. The centre of Trollius expertise in Europe is in Bristol, UK at K-par Systems Ltd. An electronic mail discussion group is operated from OSU. To participate or to make other inquiries or documentation requests, write to trollius@tbag.osc.edu or

Trollius Project
Research Computing
The Ohio State University
1224 Kinnear Rd.
Columbus, OH 43212

Epilogue

The goal of Trollius would be to create or influence a standard that would increase productivity and generate enough comfortable software to capture the more conservative supercomputer user. If accomplished, the objective would sell hardware, make money, and facilitate more science.