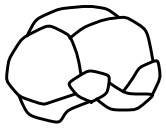


Contact

Research Computing
The Ohio State University
1224 Kinnear Road
Columbus, OH 43212

Consulting

Phone: (614) 292 8492
Internet: trollius@tbag.osc.edu

Copyright

This document is protected by copyright.
Author: GDB

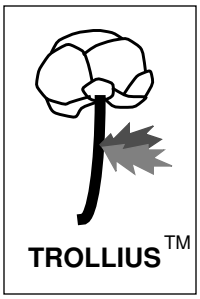
This document is based in part on the 2.0 edition of *Trollius User's Reference*, dated July 5, 1990, Copyright 1990 The Ohio State University and Cornell Research Foundation.
Authors: RYL/MB

© Copyright 1992 The Ohio State University

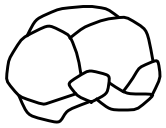
Trademarks

Trollius is a trademark of The Ohio State University and the Cornell Research Foundation.

UNIX is a trademark of AT&T.



Trollius Command Reference



Next we ship a route table to node 100 and node 0 with the trout command. Trout takes the destination nodeids as an argument, using the same syntax as all other Trollius commands.

```
% trout -v n100,0 -c route.1x4
```

We will now begin to boot the ITB nodes in earnest. This is done with the vboot command. Vboot drives three key features of the remote booting system: loading programs from the filesystem to a remote ITB node, booting a remote ITB node from its ITB node neighbour, and loading programs on a remote ITB node from its ITB neighbour. The remote ITB node being booted is called the target node and the remote ITB neighbour node doing the booting is called the staging node.

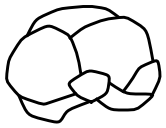
We have already booted the first ITB node. We now use it as a staging node for its neighbour, node 1. To prepare the staging node to boot its neighbour, we load the programs that constitute the operating system (from the conf.ivb file) from the filesystem onto the staging node.

```
% vboot -v -ax -al n0
```

We wish to boot the remaining ITB nodes in ascending order, using the previously booted node as the staging node. For each node, we will boot it and then load the operating system programs onto it from the staging node. Each newly booted node must also be sent a route table before we can proceed to the next node. Again we use the trout command, as above. The target node identifier is given to vboot with the -n option. The link number is specified with the -L option. According to the boot schema, we must boot across link 2, as this link is used to travel from lower numbered nodes to higher numbered nodes.

```
% vboot -v -bx -bl -sx -sl -n 1 -L 2 n0
% trout -v n1 -c route.1x4
% vboot -v -bx -bl -sx -sl -n 2 -L 2 n1
% trout -v n2 -c route.1x4
% vboot -v -bx -bl -sx -sl -n 3 -L 2 n2
% trout -v n3 -c route.1x4
```

This is a simple but slow and cumbersome way of booting. Consult the relevant manual pages for a more complete explanation.



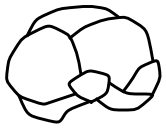
```
/*
 * bnail.1x4
 * 1D grid of 4 ITB nodes connected to the OTB
 * node via devlink
 */
/*
 * OTB node
 */
TNODE = <100, 0>;

/*
 * ITB nodes
 */
TNODE = <0, 1> <1, 1> <2, 1> <3, 1>;

/*
 * OTB link connections
 */
TLINK = <100, 0, "/dev/link", 0, 1,,> <0, 1,, 100,
     0, "/dev/link",>;

/*
 * ITB grid connections
 */
TLINK = <0, 2,, 1, 1,,> <1, 1,, 0, 2,,>;
TLINK = <1, 2,, 2, 1,,> <2, 1,, 1, 2,,>;
TLINK = <2, 2,, 3, 1,,> <3, 1,, 2, 2,,>;
```

The first node to boot must be the ITB node that is the neighbour of the OTB node. The boot schema tells us that this is node 0 and that /dev/link is the name of the OTB node's link that connects to node 0. We use sboot with the -f option to specify the link name. The -v option reports each step in the sboot procedure.

```
% sboot -v -f /dev/link
```

The host OTB node is then booted with the hboot command. The -T option ensures that the communication processes use the correct link device. The value 0x80000000 is the fixed synchronization event associated with Trollius link 0.

```
% hboot -v -T "/dev/link 0x80000000"
```



sboot	Boot OTB connected ITB node.
hboot	Boot OTB node.
vboot	Boot remote ITB node.
trout	Ship route table to a node.

Low Level Booting

The spread command runs lower level (command and toolkit) programs that perform specific parts of the booting procedure. Normally, the user will only need to use spread. In some special circumstances, when variations in the normal booting procedure not controllable with spread options are desired, the user may wish to manually boot the system. By running the low level tools and commands, the user can select options that tailor the boot to his/her needs.

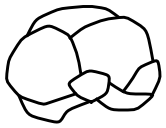
The low level boot commands and tools are sboot, hboot, vboot and trout. Sboot boots an ITB node which is a neighbour of the OTB node. Hboot boots the host OTB node. Vboot boots an ITB node that is not a neighbour of the OTB node. Trout ships a route table to a node.

Process Schema

The sboot and hboot tools and vboot command both read a node configuration file called a process schema. The process schema contains a list of programs and runtime arguments that will constitute the operating system on a node. Different nodes may require different process schemata. Filenames have the form conf.nodetype. The default process schema filename for sboot is conf.isb, for hboot is conf.otb and for vboot is conf.ivb. Just as the user can create custom boot schemata, he/she can create custom process schemata. They make it easy to reconfigure Trollius at the program level. For a complete description of the process schema grammar, see the conf manual page.

An ITB process schema may be frozen for speedier booting. Sboot's -Z option freezes a process schema into a binary file that cannot be altered. Configuration flexibility is sacrificed for speed. The default frozen process schema names for sboot and vboot are conf.isbz and conf.ivbz respectively.

To manually boot a multicomputer, first consult the boot schema. This file specifies the identifiers and types of nodes, and their interconnections. As with spread, we assume that the hardware is configured to exactly match the boot schema. We also assume that a route file corresponding to the boot schema exists. An example boot schema, describing a one dimensional array of four ITB nodes connected to one OTB node, is presented below.



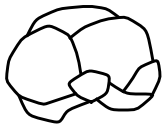
Map and Rasm

The map topology tool inputs a boot schema and outputs a route table file. It does not become a Trollius process, so it may be run without Trollius on any computer. Map is only used once for each new boot schema. If the boot schema file name has the form bnail.suffix, then the route table filename will have the form route.suffix. The -c option introduces the boot schema filename. See the map manual page for a description of other options.

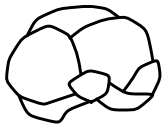
```
% map -c bnail.topology
```

Map can build a complete route table from only TNODE and TLINK tuples. With only this information, it generates every non-neighbour route. An ambitious boot schema author can specify a TROUTE tuple for every non-neighbour route and the intelligence of map will not be needed. Trollius provides a tool, called rasm, that simply translates the contents of the boot schema into a route table.

```
% rasm -c bnail.topology
```



```
/*
 * OTB connection
 */
TLINK = <100, 0, "/dev/link", 0, 1,,>
     <0, 1,, 100, 0, "/dev/link",>;

/*
 * ITB grid links
 */
TLINK = <0, 2,, 1, 1,,> <1, 1,, 0, 2,,>;
TLINK = <1, 2,, 2, 1,,> <2, 1,, 1, 2,,>;
TLINK = <2, 2,, 3, 1,,> <3, 1,, 2, 2,,>;
TLINK = <4, 2,, 5, 1,,> <5, 1,, 4, 2,,>;
TLINK = <5, 2,, 6, 1,,> <6, 1,, 5, 2,,>;
TLINK = <6, 2,, 7, 1,,> <7, 1,, 6, 2,,>;
TLINK = <0, 3,, 4, 0,, 2> <4, 0,, 0, 3,, 2>;
TLINK = <1, 3,, 5, 0,,> <5, 0,, 1, 3,,>;
TLINK = <2, 3,, 6, 0,,> <6, 0,, 2, 3,,>;
TLINK = <3, 3,, 7, 0,,> <7, 0,, 3, 3,,>;

/*
 * route table entries
 */
TROUTE = <3, 0, 1,>;
TROUTE = <2, 0, 1,>;

/*
 * priority generation of routes
 */
TORDER = <7, 4, 10>;

/*
 * vendor specific node assignments
 */
TSOLDER = <100,,>;
TSOLDER = <0,, "Acme-0"> <1,, "Acme-1">;
TSOLDER = <2,, "Acme-2"> <3,, "Acme-3">;
TSOLDER = <4,, "Acme-4"> <5,, "Acme-5">;
TSOLDER = <6,, "Acme-6"> <7,, "Acme-7">;
```



where nodeid is the node identifier given in the TNODE tuple, hostname is the machine name of the OTB host node[1], and sysnode is the vendor specific name for the node.

If the system on which we wish to establish the 2x4 grid topology has vendor specific node identifiers of the form Acme-x, our TSOLDER section might look like this:

```
TSOLDER = <100,,>;
TSOLDER = <0,, "Acme-0"> <1,, "Acme-1">;
TSOLDER = <2,, "Acme-2"> <3,, "Acme-3">;
TSOLDER = <4,, "Acme-4"> <5,, "Acme-5">;
TSOLDER = <6,, "Acme-6"> <7,, "Acme-7">;
```

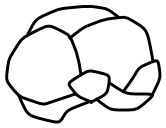
The OTB node must be included even though it does not have a vendor specific name.

Complete Boot Schema

Our completed boot schema for a 2x4 grid is:

```
/*
 * bnail.2x4
 * 2D grid of eight ITB nodes connected to
 * the OTB node via /dev/link
 */
/*
 * default link weights
 */
TWEIGHT = <1, 1, 1, 1>;

/*
* OTB node (numbered 100)
*/
TNODE = <100, 0>;

/*
 * ITB nodes (numbered 0-7)
 */
TNODE = <0, 1> <1, 1> <2, 1> <3, 1>;
TNODE = <4, 1> <5, 1> <6, 1> <7, 1>;
```

1. The name of the OTB host node will never be needed in Trollius 2.2.



The last step of the route, from node 1 to node 0 by link 1, does not need to be included, as this information is already present in the TLINK tuples.

TORDER Tuple

The TORDER tuple provides a less specific way to influence route determination by the map tool. Instead of directly specifying a route, TORDER tells map to give priority to generating the specified route. Routes specified by TORDER will be generated before routes not included in TORDER tuples. They will be generated in the order they are listed in the TORDER section.

Due to traffic management efforts, the routing tables created by map depend on the order in which routes are determined. Early in the mapping process, when no links have accumulated a heavy weight, the paths created by map are likely to be as direct as possible. Later in the mapping process, extra links may be added to a path in order to bypass a link with a heavy weight. If you expect a path will be heavily used, it is advantageous to include it in the TORDER section, specifying a heavy link weight. This will discourage other routes from incorporating these links. It will also ensure that the route length will be as short as possible.

The syntax for the TORDER tuple is:

TORDER = <<src_nodeid>**,** <dest_nodeid>**,** [<this_order_wt>]**>;**

where this_order_wt overrides the default weight given in the TWEIGHT tuple, for all links along the route generated by map. For our 2x4 grid example, we will have the path from node 7 to node 4 determined first. The TORDER tuple is:

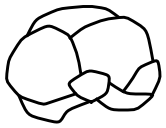
```
TORDER = <7, 4, 10>;
```

Assigning a link weight (10) greater than the default weight (1) discourages map from routing other messages through the links used for this route.

TSOLDER Tuple

TSOLDER tuples are used by systems with programmable links, to establish a link configuration other than the existing configuration. Solder uses the TSOLDER tuples to associate the Trollius node identifiers given in the TNODE tuples with actual nodes, and then configures (programs) the links so that nodes identified as adjacent in the TNODE tuples are directly connected. The syntax is:

TSOLDER = <<nodeid>**,** [<hostname>]**,** [<sysnode>]**>;**



```
TLINK = <1, 2,, 2, 1,,> <2, 1,, 1, 2,,>;
TLINK = <2, 2,, 3, 1,,> <3, 1,, 2, 2,,>;
TLINK = <4, 2,, 5, 1,,> <5, 1,, 4, 2,,>;
TLINK = <5, 2,, 6, 1,,> <6, 1,, 5, 2,,>;
TLINK = <6, 2,, 7, 1,,> <7, 1,, 6, 2,,>;
TLINK = <0, 3,, 4, 0,, 2> <4, 0,, 0, 3,, 2>;
TLINK = <1, 3,, 5, 0,,> <5, 0,, 1, 3,,>;
TLINK = <2, 3,, 6, 0,,> <6, 0,, 2, 3,,>;
TLINK = <3, 3,, 7, 0,,> <7, 0,, 3, 3,,>;
```

TROUTE Tuple

The TROUTE tuple is used to specify an entry in a node's routing table. A routing table entry simply states that a message at the local node, whose final destination is a stated node, must travel out a stated link. It is of no concern to the local node where the message originated, or what other nodes lie along the path to the destination node.

TROUTE tuples will be incorporated directly by map, without being changed in any way. All or part of a complete path between two nodes can be specified with multiple TROUTE tuples. If desired, complete routing tables can be specified.

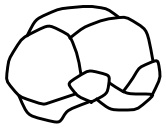
The syntax for the TROUTE tuple is:

TROUTE = <<any_nodeid>**,** <dest_nodeid>**,** <outlink>, [<outlink2>]**>;**

This specifies that a message that originates at, or passes though, any_nodeid on its way to dest_nodeid will be sent out on outlink, which is one of the links available from any_nodeid. A second route, represented by outlink2, is guaranteed to be the shortest, non-reconvergent route. It is used by multicast messages.

As an example, we will set up a route from node 3 to node 0. As this is a multi-hop route, it is necessary to choose the path that messages will travel, and to write a TROUTE tuple for each node on the path. Our chosen path is across the row, from node 3 to 2 to 1 to 0. The TLINK tuples have established that link 1 is used within the row to link higher numbered nodes to lower numbered nodes. Our TROUTE tuples are:

```
TROUTE = <3, 0, 1,>;
TROUTE = <2, 0, 1,>;
```



We will conceptualize our 2x4 grid as follows. Nodes 0-3 form one row; nodes 4-7 form the second row. Links 1 and 2 will communicate within the rows; links 0 and 3 will communicate between the rows. The OTB node will be connected to ITB node 0. Tuples describing the linking of nodes 0 and 1 (within the row) using links 1 and 2 are:

```
TLINK = <0, 2,, 1, 1,,> <1, 1,, 0, 2,,>;
```

Since a single tuple is unidirectional, we almost always write two complementary tuples. The first tuple describes the links used to go from node 0 to node 1 (out link 2, in link 1). The second tuple describes the links used to go in the opposite direction, from node 1 to node 0 (out link 1, in link 2). We have left the this_link_wt field empty. The default link weight will be used. By leaving the this_link_wt field empty, we ensure that any change made later to the default link weight will automatically be applied towards the TLINK tuple.

To link nodes 0 and 4 (between rows), we use links 3 and 0. We will give this link a weight greater than the default, in order to decrease traffic on the link.

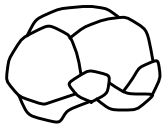
```
TLINK = <0, 3,, 4, 0,, 2> <4, 0,, 0, 3,, 2>;
```

To link the OTB node (100) and node 0, we need to name the OTB communication device. In the example system, the name is devlink. The numbers chosen to identify logical links on the OTB should be in the range 0 to LINKPERNODE - 1. Our OTB TLINK tuple line is thus:

```
TLINK = <100, 0, "/dev/link", 0, 1,,>
     <0, 1,, 100, 0, "/dev/link",>;
```

The complete TLINK section is:

```
/*
 * OTB connection
 */
TLINK = <100, 0, "/dev/link", 0, 1,,>
     <0, 1,, 100, 0, "/dev/link",>;

/*
 * ITB grid connections
 */
TLINK = <0, 2,, 1, 1,,> <1, 1,, 0, 2,,>;
```



Node types are interpreted as a collection of bits. Therefore, a node type that combines two capabilities is specified by adding the node types for the individual node capabilities. An ITB node with an attached disk thus has the node type 9.

In our 2x4 grid, we have 1 OTB node and 8 ITB nodes:

```
/*
 * OTB node (numbered 100)
 */
TNODE = <100, 0>;

/*
 * ITB nodes (numbered 0-7)
 */
TNODE = <0, 1> <1, 1> <2, 1> <3, 1>;
TNODE = <4, 1> <5, 1> <6, 1> <7, 1>;
```

Note that we have used multiple tuples, and have separated OTB and ITB node information for the sake of clarity.

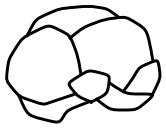
TLINK Tuple

The TLINK tuple is used to specify unidirectional physical links between nodes. This is where a specific topology is truly incorporated. The syntax for the TLINK tuple line is:

TLINK = <<from_nodeid>, <outlink>, [<oname>], <to_nodeid>, <inlink>, [<iname>], [<this_link_wt>]**>;**

The TLINK tuple states that a physical link exists between the nodes from_nodeid and to_nodeid, and specifies the links used to travel between them. You leave from_nodeid via outlink (which is one of from_nodeid's links), and enter to_nodeid via inlink (which is one of to_nodeid's links). This_link_weight is an arbitrary integer. It should be chosen in relation to the default link_wt of the TWEIGHT tuple. The link numbers outlink and inlink are logical link numbers. On ITB nodes, they are also the physical link numbers. On OTB nodes, the links must be named. Oname and iname give the name of the physical link that corresponds to the logical link preceding it in the tuple.

The maximum number of links per node is set by LINKPERNODE in the internal header file nail.h. The factory configured value is 4.



The fields of the TWEIGHT tuple assign default weights as described below.

<link_wt>	to links in TLINK tuples
<host_wt>	to paths to and from the origin host node
<order_wt>	to paths defined in TORDER tuples
<mcast_wt>	to paths established by map while creating a fully connected topology for multicasting
<hop_wt>	to every hop of every path

The weights chosen for each field can be any arbitrary integer. It is the relative weight that is important. If a weight is not specified for one of the above fields, a default weight of 100 will be used for that field. If the TWEIGHT tuple is absent, default weights of 100 will be assigned to all of the fields, except hop_wt, which is set to 0. Setting hop_wt to a large relative number forces shortest path routes. The hop_wt does not accumulate on the chosen path like the other weights.

For our example 2x4 grid, we will have the default weights be minimum link weights, and will specify a higher weight when we wish to avoid traffic on a link. We will use default weights of 1. This makes it very difficult to accidentally assign a weight smaller than the default weight.

```
TWEIGHT = <1, 1, 1, 1, 0>;
```

TNODE Tuple

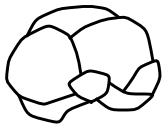
The TNODE tuple is used to list all the nodes in the multicomputer, and to specify their types. The syntax for the TNODE tuple is:

TNODE = <<nodeid>**,** <node_type>**>;**

Nodeid is an arbitrary integer used to identify a node. The node identifiers chosen in the TNODE tuple will be used to identify nodes in Trollius user interface commands and in C programming under Trollius.

Node_type identifies the kind of node. Node types are:

0	OTB (host) node
1	ITB node
4	node not available to the user
8	node with attached disk
16	node with attached video display



TROUTE	specifies an entry in one node's routing table
TORDER	prioritizes the generation of an optimal path between the specified nodes
TSOLDER	associates TNODE node identifiers with actual nodes

A boot schema must contain TNODE and TLINK tuples. This is the minimum amount of data needed to generate a topology. The TWEIGHT, TROUTE, and TORDER tuples allow a varying degree of user control over the routing tables generated by the map tool. TSOLDER is only used by systems with programmable links. In the case of duplication or conflict of information between various tuples, the information in the first tuple to appear in the boot schema is used. Subsequent tuples do not overwrite this information. Another restriction is that a TNODE tuple must appear before any other tuple using its nodeid.

We will now describe the tuple types in depth, writing the appropriate tuples for a 2x4 grid as we go along. System specific information, such as link numbers, device driver names, and vendor specific node identifiers will be used. Keep in mind that the values used in this example boot schema may differ from those for your system. System specific information can be found in existing boot schemata, or may be obtained from the hardware vendor.

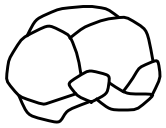
TWEIGHT Tuple

The TWEIGHT tuple is used by map when generating route tables from the boot schema. Map attempts to avoid congestion on the links by keeping track of the total weight accumulated on a given link. Every time a link is used as a part of a route, a certain weight is added to that link's total weight. If the added weight of all the links on a route is lower than for alternate routes, that route will be selected by map for inclusion in the routing tables.

The default link weights used in this optimization are established in the TWEIGHT tuple. As we will see, these default weights can be overridden for specific links in the TLINK and TORDER tuples. The syntax for the TWEIGHT tuple line is:

```
TWEIGHT = <[<link_wt>], [<host_wt>], [<order_wt>],
      [<mcast_wt>], [<hop_wt>]>;
```

As for all tuples, the TWEIGHT tuple consists of fields separated by commas and bounded by angle brackets. The tuple line consists of a tuple name (TWEIGHT, in this case), **=**, and one or more tuples of the corresponding type, with a terminating semicolon. The use of brackets ([ ]) indicates that the field can be left empty, although the separating commas are mandatory.



map -c <bnail> Compile route table to match boot schema.
rasm -c <bnail> Assemble route table to match boot schema.

```
TWEIGHT = <<[link_wt]>, [<host_wt>], [<order_wt>],
                        [<mcast_wt>], [<hop_wt>]>;
TNODE = <<nodeid>, <node_type>>;
TLINK = <<from_nodeid>, <outlink>, [<oname>], <to_nodeid>,
                        <inlink>, [<iname>], [<this_link_wt>]>;
TROUTE = <<src_nodeid>, <dest_nodeid>, <outlink>, [<outlink2>]>;
TORDER = <<src_nodeid>, <dest_nodeid>, [<this_order_wt>]>;
TSOLDER = <<nodeid>, [<hostname>], [<sysnode>]>;
```

Writing a Boot Schema

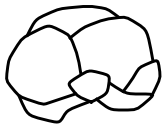
The topology of a multicomputer is established in the boot schema. The boot schema specifies the identifiers and types of nodes, and their interconnections. It may also contain information on message routing and on vendor specific nodeids. The boot schema is used by map and rasm to generate the route tables, by solder to configure the links, and by spread when booting the operating system.

A variety of boot schemata describing different topologies should already be available for a given installation. These files are generally found in the directory $TROLLIUSHOME/boot. Boot schema files are named bnail, suffixed with a term describing the topology of the boot schema. For example, bnail.2x4 describes a 2 by 4 grid topology. Some users may wish to write their own boot schema. They may wish to use a topology for which no boot schema currently exists, to add routing information to an existing boot schema, or to adapt a boot schema to a different system. In this section, we will describe how to write a boot schema, using the 2 by 4 grid topology as an example.

NaIL

Boot schemata are written in Node and Interconnect Language (NaIL). This language supports comments bounded by /* and */, and six types of topology tuples. Tuples are sets of information used to define certain aspects of topology and routing. The different tuples types are described below.

TWEIGHT	specifies default weights used by map when generating optimal paths
TNODE	specifies node identifiers and types
TLINK	lists physical links between nodes



loadgo -t	Run and trace ITB program.
ttrace	Unload trace data.
trstate	Monitor trace collection.
trade	Convert trace data to ParaGraph format.

Trace Collection

The execution of a parallel program can be traced and visualized for the purpose of debugging or performance analysis. In particular, data are gathered on message passing and the busy/idle times of each processor. Trace collection is enabled in an ITB application process by using the -t option with the loadgo command.

```
% loadgo -t C appl
```

Processes collect trace data in runtime buffers and then move the data to a central trace daemon, traced, on each node as the buffers fill. The user must ensure that traced is running on all nodes that will run trace producing processes. Once an application has completed (or even while it is running) trace data can be collected from each node into files on the host node. This is one function of the ttrace command.

```
% ttrace n0-3 appl.dat
```

The status of trace collection on any node can be monitored with the trstate command.

```
% trstate n0-3
```

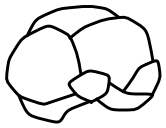
The raw Trollius trace data must be converted into a format readable by a visualization tool. The trade command is provided to convert the data to ParaGraph format. ParaGraph is a performance visualization system available from netlib (netlib@ornl.gov) at Oak Ridge National Laboratories.[1]

```
% trade appl.dat -o appl.trf
```

Caveats

While the trace generation and collection system has been designed and implemented to run on ITB and OTB nodes, clock synchronization has not been implemented between ITB and OTB nodes and the user must therefore confine tracing to ITB nodes. Also, the current ITB clock synchronization implementation requires that there only be one ITB/OTB link.

1. “Visualizing the Performance of Parallel Programs”, Michael T. Heath and Jennifer Etheridge, IEEE Software, Sept. 91, Vol 8, No. 5, pp. 29-39



-S option does the same thing for all file descriptors. With no options, tfile prints the current working directory of the remote filesystem. The working directory is changed by giving a new pathname to tfile. Tcd is a command wrapper for tfile h.

```
% tfile -s 4
% tcd /example/data
```

Mstate

The mstate command monitors the status of memory allocation. In its default display mode it prints information on allocated blocks under four headings.

```
% mstate n0
```

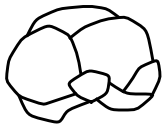
NODE	nodeid of memory block
PID	owner pid of memory block
ADDRESS	base address of memory block
SIZE	size of memory block

The -f options switches to the alternate display mode, where information is printed on unallocated or free memory blocks. The display is unchanged, but the message “free” is always printed under the PID heading.

Drain

In extreme debugging situations, the drain command can copy memory from an ITB node into a file on the OTB node. The arguments to the drain command are the target nodeid, the memory block base address, the length of the memory block and the filename that will hold the memory block on the OTB node.

```
% drain n0 0x80000000 0x1000 on-chip
```



Fstate and Tfile

```
% fstate h
```

The remote filesystem[1] has monitoring and control commands. Fstate prints one line of status information for each open file descriptor under the following headings:

FD	global file descriptor handle, not the same as the handle seen by the client process
FLAGS	open flags and status flags (see below)
FLOW	total amount of I/O in bytes since opening
CLIENT	nodeid and event of last client process
NAME	filename

The open/status flags are single character mnemonics.

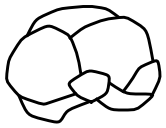
R	open for read
W	open for write
S	Cubix single access method
M	Cubix multiple access method
L	locked active
A	active, currently open in the underlying filesystem
I	inactive, currently closed in the underlying filesystem

Fstate has an alternate display mode for Cubix files only. The -c option prints information under the following headings:

FD	as above
FLAGS	as above
SYNC'D	number of client synchronized on a Cubix method file request over the total number of clients
REQ	current Cubix method file request
NAME	as above

The tfile command has two features. When Cubix programs fail, they can leave partially synchronized open file descriptors. The -s option cleans up and closes a specific file descriptor (even non-Cubix descriptors) while the

1. In many multicomputers, only the OTB node has a filesystem so only the OTB node has a Trollius remote filesystem service. In this situation, the only valid nodeids for the fstate and tfile commands are the OTB nodeid and h, a shorthand for the OTB nodeid.



For each given nodeid, bfstate prints one line of information for every buffered message on the node. The -l option adds some information at the end on current buffer parameters. The bfstate headings are described below.

NODE	message resides on nodeid
DEST	message is bound for destination nodeid
EVENT	synchronization event at destination nodeid
TYPE	synchronization message type at destination nodeid
LENGTH	message length, always less than MAXNMSGLEN
STATE	“blocked” or “waiting”

A blocked message resides with a buffer worker that is blocked trying to forward the message along its route. A waiting message resides in the main buffer pool. It cannot be forwarded until a buffer worker becomes free.

```
% bfstate n0
% buffer n0 -s 0x100000
```

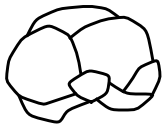
The user can control the maximum number of buffer worker processes (-b) and the maximum size of the main buffer pool (-s) with the buffer command. The -S option, also wrapped as the sweep command, cleans out all buffered messages on the specified nodes. This is useful and necessary after an unsuccessful program run and before subsequent runs.

Doom

Doom is the command level interface to signal delivery. It handles multiple nodes and various process selection methods in the same way as the state command. With no other options, doom sends a SIGUDIE signal to the selected processes. Unfortunately, the user cannot specify signal mnemonics and must give the actual signal number instead. These are listed below.

4	(SIGUDIE) terminate
5	(SIGARREST) suspend execution
6	(SIGRELEASE) resume execution
7	(SIGA) reserved for user
8	(SIGB) reserved for user

Doom does not guarantee immediate signal delivery. Signals will be delivered on the selected process’s next or current kernel request.



KSTATE	kernel execution status, see below
PROGRAM	load module file name, sometimes unknown, only ITB

The kernel execution status is shown as a one or two letter code followed by the last or current blocked message event and message type. The status codes are described below.

RM	ready controlled process; completed ksend() or krecv() request; also seen as **R**
RS	ready; completed kstate() request
RE	ready; encountered error during kernel request
RA	ready; completed kattach() request
R	ready; completed kdetach(), kdoom(), ksend() or krecv() request
BS	blocked sending a message to an event/type
BR	blocked receiving a message from an event/type
A	active; not under kernel control
F	free; not attached to the kernel (either never attached or has detached)

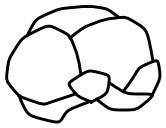
If no user processes are controlled processes (non-zero kernel priority), then the state command should rarely see a user process in a state beginning with R. With message buffering, BS is also uncommon. The most common states are A and BR.

In alternate display mode, triggered by the -l option, state prints information under the following headings:

NODE	process nodeid
INDEX	process index
STACK	base address of stack segment
STATIC	base address of static data segment
LOADPT	base address of load module, including text segment
SIGS	undelivered signals
PROGRAM	load module file name, sometimes unknown, only ITB

Bfstate and Buffer

A receiving process is usually debugged with the state command, but a sending process emits a message and returns to an active state very quickly due to the presence of buffers. The bfstate command is provided to examine buffered messages.



state <nodeid> <process>	Print status of remote process.
doom <nodeid> <process>	Send signal to remote process.
bfstate <nodeid>	Print message buffer status.
fstate <nodeid>	Print remote filesystem status.
tfile <nodeid>	Control remote filesystem.
mstate <nodeid>	Print ITB memory allocation status.
buffer <nodeid>	Control message buffers.
drain <nodeid> <address> <length> <path>	Copy ITB memory into file.

Monitoring and Controlling Processes

Monitoring a process's execution state and monitoring messages in transit are major aids in developing multicomputer applications. These features help debug process synchronization and data transfer, the added dimension of parallel programming.

State

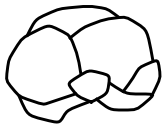
To gather information on ITB processes, use the state command and specify one or more ITB nodeids. There are different ways to select processes for status reporting. All of these methods only affect remote processes, usually started by loadgo. The numerical <list> follows the same syntax as nodeid lists.

%<list>	Select process indices in <list>.
#<list>	Select processes blocked on events in <list>.
@<loadpt>	Select processes with load module addresses in <list>.
p<list>	Select processes with pids in <list>.
-t	Select all system processes.
-g	Select all processes.
<none>	Select all user processes.

```
% state n0
```

OTB processes can also be monitored, but only with the pid selector. In its default display mode, state prints information under the following headings.

NODE	process nodeid
INDEX	process index, generally easier to use than pid but effectively equivalent
PID	process identifier
KPRI	kernel priority, set by kinit()



loadgo <nodeid> <load_module> **--** <args>
Load and execute ITB program.
<load_module> <args> **&** Execute OTB program.

Executing Programs

The loadgo command loads a load module from the local OTB node's file-system onto the specified ITB node and creates a remote process. Multiple ITB nodes, including a multicast node can be specified and one process will be created on each node. Multiple load modules can be specified and one process will be created for each load module on each node.

One set of command line arguments can be given with the -- option. Everything after the -- on the loadgo command line is not recognized by loadgo but instead is passed as command line arguments to every created process, in the C standard argc/argv format.

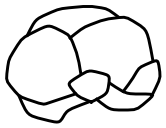
```
% loadgo C -v appl -- -v
```

By default, loadgo allocates 8 kilobytes of stack space for every new process. The stack size is fixed during execution. Use the -s option to specify a different stack size.

Loadgo returns control immediately after the last process is created. This presents a problem for programs which output to the terminal and a major problem for terminal input. The -w option forces loadgo to wait until all processes have completed before returning control to the shell. This allows a terminal interactive program to function correctly but interactive access through the shell is suspended temporarily. Standard input and output (but not standard error) may be redirected to files by the -i and -o options.

Trollius OTB programs are executed like any UNIX program. The process may be placed in the background or foreground.

```
% appl -v &
```



Tkill

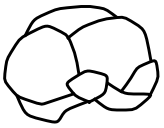
To terminate Trollius on the OTB node and release ITB hardware resources, use the tkill command. It is not a bad idea to place tkill in your .logout file. To reboot Trollius after a system failure, execute tkill followed by spread.

```
% tkill
```

If tping or any other Trollius command does not operate correctly, it almost always means that the software on one or more nodes has crashed. It is usually futile to issue further commands at this point. Reboot the multicomputer with tkill and spread.

Resource Management

Resource management is not part of second generation Trollius. Trollius 2 attempts to integrate with the hardware vendor's resource management software, if it exists. The booting procedure may be augmented by vendor specific allocation and deallocation commands. Consult the trollius(l) manual page for specific details about how Trollius integrates into a specific vendor's resource management system.



solder –v	Configure multicomputer hardware.
spread –v	Boot Trollius onto a multicomputer.
tping <nodeid>	Check communication to given node.
tkill	Terminate Trollius session.

Booting the Operating System

Trollius is prepared for a configurable multicomputer. The physical computers, ITB and OTB, which constitute the total hardware resource, may be linked in arbitrary ways to produce different interconnection topologies. Different topologies are useful to mirror the different geometry of various applications. The topological capabilities of any specific multicomputer may be limited, but Trollius topology is unlimited.

Different interconnection topologies affect the way Trollius boots the multicomputer and routes messages within it. Thus, there is a need to explicitly state the topology of the actual hardware to be booted. In Trollius the boot schema (ASCII file) serves this purpose. See *Writing a Boot Schema* for a discussion of the grammar and capabilities of a boot schema.

Solder

At boot time, Trollius assumes that the hardware configuration exactly matches the description in the boot schema. This is a very important prerequisite. A machine specific implementation of the solder tool may be provided to achieve this configuration. If solder is not present, it is up to the user to match the hardware and boot schema before booting. The -c option of solder specifies the filename containing the boot schema.

```
% solder –v –c bnailgrid.4x4
```

Spread

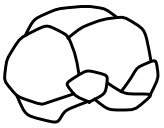
The spread command boots Trollius onto the multicomputer. The -v option prints a message before each booting step is attempted. The specific boot schema provided to solder must also be given to spread.

```
% spread –v –c bnailgrid.4x4
```

Tping

Hands-on control and monitoring give Trollius its flavour. The simplest command, tping, is a confidence building validity check that begins to dispel the black box nature of multiple ITB nodes. Tping echoes a message to a destination node, or a multicast destination.

```
% tping n0
% tping C
```



tcc –o appl appl.c typical ITB compilation
cc –I$TROLLIUSHOME/h –o appl appl.c –L$TROLLIUSHOME/lib –lt
typical OTB compilation

Compilation

A Trollius programmer compiles software for two types of processor targets, ITB and OTB. The OTB target is simply the local environment. Libraries are linked and header files are included in the same manner as any third party, locally installed software. Trollius makes no alterations to the local development environment.

```
% cc –I$TROLLIUSHOME/h –o appl appl.c
     –L$TROLLIUSHOME/lib –lt
```

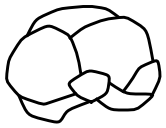
Notice that the basic services library, libt, is explicitly linked. These verbose command line options can become tedious and a simple alias or shell script is recommended, but not provided.

ITB objects and executables are built with a cross compiler. The compiler itself runs on the local OTB computer but produces code for an ITB computer. The usage of the cross compiler is similar to the native UNIX (OTB) compiler. The name will be different and if the cross compiler is configurable for different runtime environments (as any good cross compiler should be) then Trollius libt will be implicitly linked and Trollius header file directories will be implicitly searched. The Logical Systems Transputer Toolset (LSTT) contains a cross compiler that is invoked as follows:

```
% tcc –o appl appl.c
```

Take care not to confuse ITB objects and executables with OTB objects and executables, which may have the same name. Both the ITB and OTB compiler invocations shown above produce an object file called appl.o and an executable called appl.

Less essential Trollius libraries must always be linked explicitly. The two user libraries for Trollius remote services are libtrillium and libbuckeye. The math library, libm, is part of the compiler suite, not the operating system, and is also usually linked explicitly.



Some (but not all) Trollius commands allow you to specify all ITB nodes on the command line as C. The fstate command requires you to specify the OTB nodeid. This can be done as shown above using the nodeid learned from the boot schema or with the simple mnemonic, h.

Pid Mnemonics

You will also need to identify processes. Processes can be identified in several ways. All process oriented commands will accept p<list> where <list> is a pid or list of pids. Example pid specifications are:

```
p5158
p5158,5160,5200-5210
```

Most ITB node pids can be learned with the state command. OTB node pids can be learned with the UNIX ps(1) command.

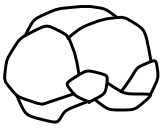
On-line Help

To print a brief summary of the syntax and options of any Trollius command level program, execute the command with the -h option.

```
% spread -h
```

Detailed information on each command (and most programming functions) is available from on-line manual pages. They are an important supplementary reference to this document. You may have to append or create the MANPATH environment variable to enable the man command to find the Trollius manual pages.

```
% man spread
```



TROLLIUSHOME	Trollius installation directory
LSTTHOME	LSTT installation directory
TROLLIUSBNAIL	default boot schema file name

Preparing the UNIX System

Before using Trollius programs you must establish certain environment variables and search paths for your shell. Add the following commands to your .login or .cshrc (assuming C shell) files.

```
setenv LSTTHOME <LSTT installation directory>
setenv TROLLIUSHOME <Trollius installation directory>
set path = ($path $LSTTHOME $TROLLIUSHOME)
```

The local system administrator, or the person who installed Trollius, will know the locations of the Trollius and LSTT installation directories. LSTTHOME is only used for the LSTT ITB cross compiler. Other compilers may or may not require other variables. After editing the chosen file, execute it with the shell's source command. This is not necessary on subsequent logins to the UNIX system.

```
% source <file>
```

The default boot schema file name can be set in an environment variable. You may or may not want to make the default schema a permanent part of your environment in the .login or .cshrc files.

```
% setenv TROLLIUSBNAIL bnail.2x4
```

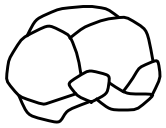
Nodeid Mnemonics

Many Trollius interface commands require you to specify a nodeid or nodeids. Nodeids are specified on the command line as n<list>, where <list> is a list of comma separated nodeids or nodeid ranges. Example nodeid specifications are:

```
n1
n1,3,5-10
```

Nodeids are established in the Trollius multicomputer plan, called a boot schema (see *Writing a Boot Schema*), and vary from system to system. During installation, your system will be set up with several common configurations of your multicomputer resource. Learn how the nodes are assigned nodeids in the default boot schema with the following command:

```
% spread -V
```



Basic Objects

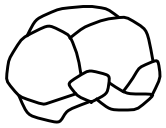
Trollius does not restrict the interconnection topology of nodes. A globally unique 32 bit identifier is given to each node (nodeid), chosen by the user. Most Trollius commands take the nodeid as a parameter. The node is the only global object in Trollius. The two other important objects are purely local to each node. They are the process and the event.

Every Trollius node has a multi-tasking, message passing local environment upon which the rest of the operating system is built. There is a small kernel at each node that manages and synchronizes any number of local processes. Each process has a locally unique 32 bit identifier (pid). Many Trollius commands operate on pids.

The event is an arbitrary 32 bit quantity and the key synchronization point for two local processes. You will see an event in three places.

- in the state of a process blocked on sending a message
- in the state of a process blocked on receiving a message
- in a network (remote node) message

Equal events synchronize. The event has an associate, subordinate object called the type which is found wherever the event is found. It is the secondary synchronization point between two local processes. It is irrelevant if the event does not synchronize. It is irrelevant if it is set to zero. Non-zero types that share a common one bit (bitwise AND is non-zero) synchronize.



munication links exist such that all computers of both types are reachable (directly or indirectly by hopping through intermediate computers). In most cases, general purpose computers, be they workstations or mainframes, are reachable via a Local Area Network (LAN). In Trollius, we refer to general purpose computers as Outside The Box computers or OTB. We recognize the following characteristics of OTB computers:

- They are shared among multiple users.
- Links to other OTB computers are shared among multiple users.
- They may control peripherals needed by a parallel application.
- They will run Trollius as an application of the native, general purpose operating system.

When running Trollius, we refer to ITB or OTB computers as nodes. Very often we make no node type distinction when describing functionality - because there is none. Unless the node type is specified, assume the function applies generally to any node.

Figure 1 represents the broad Trollius multicomputer.

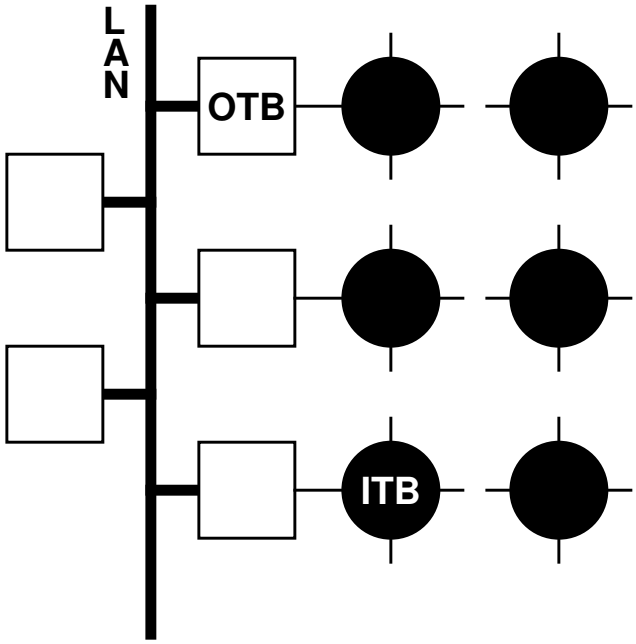
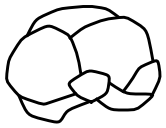


Figure 1: Trollius Multicomputer Architecture



Introduction

Trollius delivers power to the user through extensive control and monitoring functionality in a highly layered software structure. You can control system details at the lower layers or confine yourself to the comfort and safety of the higher layer. Most importantly, you can decide the balance of performance, functionality and programmer productivity that is best for a particular application. This open architecture design makes Trollius extensible, reconfigurable, and a good foundation for system experimentation.

Foundation is the word which best describes the role of Trollius in the total multicomputer software environment. It is an operating system that focuses on the basics, the nuts and bolts of parallel processing. Compilers, CASE tools for parallelism, and paradigms and abstractions of all sorts need a good foundation. Trollius was not adapted from another computing solution; it was specifically designed for multicomputer applications.

Trollius presents a broad view of the multicomputer architecture. It recognizes that any computer reachable via communication link(s) is a potential participant in a message passing application. The software presents a uniform programming environment across general purpose computers (e.g.: workstations) and dedicated highly integrated computer ensembles. The latter is usually singled out and tagged as the “parallel machine” and is packaged into an impressive looking box. In Trollius, we refer to these computers as Inside The Box computers or ITB. We recognize the following characteristics of ITB computers:

- They are dedicated to single users.
- All incoming and outgoing links are dedicated to single users.
- They are great in number.
- They are going to be used for large parallel computational problems and not the common applications that UNIX workstations do well.
- On average, they have few peripherals.
- They will run Trollius natively.

Since Trollius is not going to provide all the services of a general purpose operating system, the ITB computers must be hosted by a computer with such an operating system. The host computer will have one or more communication links to one or more ITB computers. As indicated above, other general purpose computers may participate in a multicomputer if the com-

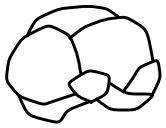


Table of Contents

Introduction 4

- Basic Objects 6

Preparing the UNIX System 7

- Nodeid Mnemonics 7
- Pid Mnemonics 8
- On-line Help 8

Compilation 9

Booting the Operating System 10

- Solder 10
- Spread 10
- Tping 10
- Tkill 11
- Resource Management 11

Executing Programs 12

Monitoring and Controlling Processes 13

- State 13
- Bfstate and Buffer 14
- Doom 15
- Fstate and Tfile 16
- Mstate 17
- Drain 17

Trace Collection 18

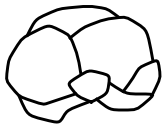
- Caveats 18

Writing a Boot Schema 19

- NaIL 19
- TWEIGHT Tuple 20
- TNODE Tuple 21
- TLINK Tuple 22
- TROUTE Tuple 24
- TORDER Tuple 25
- TSOLDER Tuple 25
- Complete Boot Schema 26
- Map and Rasm 28

Low Level Booting 29

- Process Schema 29



How to Use This Document

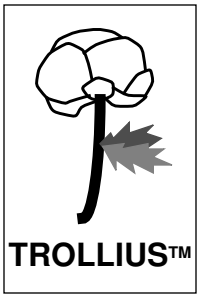
This document is an operator's reference for Trollius on multicomputers. It assumes that you have written a multicomputer program.

This document is user oriented. It does not give much insight into how the system is implemented. It does not detail every option and capability of every command. The manual pages cover all the commands in great detail and are meant to supplement this document.

Each section starts with a brief synopsis of the commands covered by the section.

We have kept the font and syntax conventions to a minimum.

`code`	This font is used for things you type on the keyboard or see printed on the screen. We use it in terminal demonstrations and tables but not in the main text.
<symbol>	This is a symbol used to abstract something you would type. We use this convention in commands and in the NaIL grammar but not for functions. Don't confuse this with the angle brackets that are actually part of the NaIL grammar.
Section	Italics are used to cross reference another section in the document.



Trollius Command Reference

Trollius is a programming environment for message passing MIMD parallel computers (multicomputers). This document covers all aspects of user operation of a UNIX based multicomputer running Trollius. Configuration of the operator's environment, control and monitoring of application programs and parallel machine configuration are discussed in the greatest detail. New users are referred to the *Trollius Tutorial Introduction in C*.

Ohio Supercomputer Center

Research Computing
The Ohio State University

Document Series 2/1 **March 29, 1992**