TOP-C (Task Oriented Parallel C/C++)

A Package for Easily Writing Parallel Applications
for both Distributed and Shared Memory Architectures
Version 2.5.0, September, 2004

Gene Cooperman

Copyright (c) 2000-2004, Gene Cooperman

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "Copying" and "GNU General Public License" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

1 'TOP-C' Copying Conditions

All of the copyright notices of this package are designed to encourage free copying and usage. This manual is copyright by Gene Cooperman. Most of the source code files of the 'TOP-C' software package contain a copyright notice similar to that below. At the current time, Some of the source files are copyright by Gene Cooperman alone and some by both Gene Cooperman and Victor Grinberg, but all files are distributed under the GNU LGPL license referred to below.


```
**********************************************************************
* Copyright (c) 2000 - 2004 Gene Cooperman <gene@ccs.neu.edu>        *
*                                                                    *
* This library is free software; you can redistribute it and/or      *
* modify it under the terms of the GNU Lesser General Public         *
* License as published by the Free Software Foundation; either       *
* version 2.1 of the License, or (at your option) any later version. *
*                                                                    *
* This library is distributed in the hope that it will be useful,    *
* but WITHOUT ANY WARRANTY; without even the implied warranty of     *
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU   *
* Lesser General Public License for more details.                    *
*                                                                    *
* You should have received a copy of the GNU Lesser General Public   *
* License along with this library (see file COPYING); if not, write  *
* to the Free Software Foundation, Inc., 59 Temple Place, Suite      *
* 330, Boston, MA 02111-1307 USA, or contact Gene Cooperman          *
* <gene@ccs.neu.edu>.                                                *
**********************************************************************
```


2 Quick Start: Installation and Test Run

> This is version 2.5.0 of Task Oriented Parallel C/C++ ('TOP-C').
> See http://www.ccs.neu.edu/home/gene/topc.html for general
> information on obtaining source, overview slides of 'TOP-C',
> previous 'TOP-C' applications, etc.
>
> Gene Cooperman
> gene@ccs.neu.edu

It has been tested and works on several workstation architectures. It provides task-oriented parallelism to the end user. The same application source code (or object code) can be executed (or linked) with any of the following communication libraries using the topcc command (a substitute for cc or gcc):

1. a distributed memory library using MPI (MPI subset, mpinu, included in distribution); or
2. a shared memory library using threads; or
3. a single-process sequential library (useful for development and debugging).

The 'TOP-C' model has been successfully used for some very large computations.

To unpack:

```
gunzip topc.tar.gz
tar xvf topc.tar
cd topc
./configure
make
```

If you are impatient, you can immediately do:

```
cd bin
make parfactor
```

and see a demonstration for factoring numbers by a Euclidean sieve. Then try: `./a.out` *YOUR_NUMBER*
Read bin/Makefile for an example of compiling a 'TOP-C' program.

If you want to use more or different slave processes, modify 'bin/procgroup'. Then, again:

```
cd bin
./a.out YOUR_NUMBER
```

And compare with your results using a single slave process:

```
./a.out --TOPC-num-slaves=1 YOUR_NUMBER
```

Or investigate what other command-line TOP-C options are available for a.out.

```
./a.out --TOPC-help
```

Other TOP-C applications besides parfactor are in the directory '../examples'. For example, if you would like to try 'parquicksort.c', then from the 'bin' directory, try:

```
make check TEST_FILE=parquicksort
```

When you write your own 'TOP-C' application, 'app.c', make sure that it contains `#include <topc.h>` at the top, and that you have a procgroup file in the current directory, and then:

```
bin/topcc --mpi -o app app.c
# or else:  bin/topc++ --mpi -o app app.cc
./app
```

'topcc' is a wrapper for your standard C compiler. It accepts all the standard compiler command line options, plus a few more. 'topc++' also exists. [NOTE: `bin/topcc --help` and `bin/topc++ --help` both work and 'doc/topcc.1' exists]

This might be a good point to download and print at least the first half of the manual, or else make it from the doc directory (`cd doc; make pdf; make ps`). Note that the table of contents may appear at the end of the documentation after the index. See Appendix A [Summary], page 34, for a list of the 'TOP-C' commands. See Appendix B [Example], page 37, for a simple 'TOP-C' example application involving only trivial parallelism. The 'examples' subdirectory provides more detailed examples. A tutorial is available from the web page, `http://www.ccs.neu.edu/home/gene/topc.html`.

Finally, if you want to install 'TOP-C' permanently, type

```
./configure                  # Install in /usr/local
./configure --prefix=$HOME  # Install in home directory
make install
```

This adds a man page, an info file (try `info topc`), etc. HTML documentation is available in the 'doc' subdirectory of the 'TOP-C' distribution. By default, 'TOP-C' will install in '/usr/local', requiring root privilege. If you configure --prefix=$HOME, 'TOP-C' will create files in '$HOME/bin', '$HOME/lib', '$HOME/include', '$HOME/man' and '$HOME/info'. If you want to make 'TOP-C' with cc instead of gcc, type:

```
env CC=cc ./configure --cache-file=/dev/null
```

CFLAGS, CXX (for topc++ and CXXFLAGS can also be set. For other configuration options, type:

```
./configure --help
```

Please write the author, Gene Cooperman (gene@ccs.neu.edu), if there are any questions.

3 Overview of 'TOP-C/C++'

> "Difficulty?" exclaimed Ford. "Difficulty? What do you mean difficulty? [The wheel is] the single simplest machine in the universe!" ...
> "All right, Mr. Wiseguy," she said, "you're so clever, you tell us what color it should be."
> from "The Restaurant at the End of the Universe"
> by Douglas Adams

'TOP-C' has been designed especially to make it easy to parallelize *existing* sequential applications. Toward this goal, 'TOP-C' emphasizes:

1. ease of use (high level task-oriented abstraction);
2. latency tolerance; and
3. architecture independence (same application code for shared and distributed memory).

A 'TOP-C' application is compiled and run using topcc (similarly to gcc) or topc++ (similarly to g++). For example, assuming a 'procgroup' file in the current directory to specify the remote hosts for the slave processes, one executes:

```
topcc --mpi parfactor.c
# or else:  topc++ --mpi parfactor.cc
./a.out
```

If a 'TOP-C' application fails to link, check for a clash of symbol names. All TOP-C symbols are of the form TOPC_*, COMM_*, MEM_*, MPI_*, MPINU_*, NO_ACTION, UPDATE, REDO, CONTINUATION, or NOTASK.

For purposes of documentation, we will standardize on an explanation of topcc. Wherever topcc is mentioned, the description is equally valid for topc++.

3.1 Programmer's Model

3.1.1 Structure of a TOP-C Program

A typical TOP-C application has the following structure:

```
#include <topc.h>
... define four callback functions for TOPC_master_slave() ...
int main( int argc, char **argv ) {
  TOPC_init( &argc, &argv );
  ...
  TOPC_master_slave( GenerateTaskInput, DoTask, CheckTaskResult,
                     UpdateSharedData )
  ...
  TOPC_finalize();
}
```

The program above is run with a master process and several slave processes operating with identical command line arguments and identical initial data (SPSD, or Single Program, Single Data). Communication among processes occurs only within the routine `TOPC_master_slave()`, during which execution can be called SPMD (Single Program, Multiple Data). At the end of `TOPC_master_slave()`, execution returns to SPSD, although the master process may contain some additional process-private data.

Chapter 9 [TOP-C Raw Interface], page 30 describes an alternative interface that is often useful for parallelizing existing sequential code. However, for new applications, the standard interface will usually be cleaner.

3.1.2 Four Callback Functions

In a 'TOP-C' application, the programmer defines four callback functions and passes control to the 'TOP-C' library through the following command.

```
TOPC_master_slave(GenerateTaskInput, DoTask, CheckTaskResult,
                  UpdateSharedData);
```

Pictorially, TOP-C arranges for the flow of control among the four callback functions as follows:

```
    (ON MASTER)      task input
 GenerateTaskInput() ---------->


task input   (ON A SLAVE)  task output
-----------> DoTask(input) ----------->


task output     (ON MASTER)                      action
-----------> CheckTaskResult(input, output) ----------->


if (action == UPDATE):
  task input, task output      (ON ALL PROCESSES)
  -----------------------> UpdateSharedData(input, output)
```

3.1.3 Task Input and Task Output Buffers

A *task input* or *task output* is an arbitrary buffer of bytes of type `(void *)` in 'TOP-C'. The task buffers are arbitrary application-defined data structures, which are opaque to 'TOP-C'. Note that in ANSI C, a void pointer is compatible with any other pointer type (such as `(struct task_input *)` and `(struct task_output *)` in the example below).

Defining Application Task Buffers

An application callback function returning a task input or task output must encapsulate it in a function, `TOPC_MSG( void *`*buf*`, int `*buf_size*` )`, which is then returned by the callback functions `GenerateTaskInput()` and `DoTask()`. `TOPC_MSG()` internally allocates a copy of *buf*, and TOP-C later frees the copy automatically. So, *buf* may be reused by the application program.

```
TOPC_BUF DoTask( struct task_input * inp ) {
  struct task_output outp;
  ...
  outp = ...;
  return TOPC_MSG( &outp, sizeof(outp) );
}
```

The principle of memory allocation in 'TOP-C' is that if an application allocates memory, then it is the responsibility of the application to free that memory. `TOPC_MSG()` has the further property of copying its buffer argument to a separate 'TOP-C' space (using a shallow copy), after which the application can free any memory it allocated. This happens automatically in the above example, since `outp` is allocated on the stack.

Marshaling (Serialization) and Heterogeneous Architectures

If a heterogeneous architecture is used, there is an issue of converting data formats or *marshaling*. This is the application's responsibility. For simple data formats (integers, floats or characters), such conversion can easily be done in an ad hoc manner. Most C compilers use the IEEE binary floating point standard, and characters are almost always encoded in eight bit ASCII representation (or possibly in a standard Unicode format). Although the byte ordering of integers is not standardized, the system calls `htonl()` and `ntohl()` are available to convert integers into 32 bit *network integers* that are portable across heterogeneous systems.

For more complicated conversions, one can consider writing one's own marshaling routines or else using a standard package for marshaling such as the '`XDR`' library (RFC 1832, eXternal Data Representation), '`IDL`' ('`Corba`'), or '`SOAP`' ('`XML`').

Marshalgen, a Package for Marshaling

For complex C++ applications, we recommend tat you try out '`Marshalgen`' package. A pointer to it is available from the TOP-C home page. Since the C++ classes to be marshaled are already defined in '`.h`' files, Marshalgen asks th user simply to annotate those files with comments (for example, deep copying vs. shallow copying fo pointers). '`Marshalgen`' also has support for such real world issues as marshalingsubclasses and templates, handling of polymorphism, etc. The 'Marshalgen' preprocessor then generates methods for a new marshaling class that know how to marshal and unmarshal the original class.

3.1.4 The 'TOP-C' Algorithm

When there is only one slave, The 'TOP-C' algorithm can be summarized by the following C code.

```
{ void *input, *output;
  TOPC_ACTION action;
  while ( (input = GenerateTaskInput()) != NOTASK ) {
     do {
       output = DoTask(input);
       action = CheckTaskResult(input, output);
     } while (action == REDO);  /* REDO not useful for only one slave */
     if (action == UPDATE) then UpdateSharedData(input, output);
  }
}
```

On a first reading, it is recommended to skip the rest of this section until having read through Section Section 4.3 [Actions], page 10.

For a better understanding of the case of multiple slaves, this simplified excerpt from the 'TOP-C' source code describes the 'TOP-C' algorithm.

```
TOPC_BUF input, output;
int num_idle_slaves = num_slaves;
TOPC_ACTION action;

while (TRUE) {
  wait_until_an_idle_slave();
  input = COMM_generate_task_input();
  if (input.data != NOTASK.data) {
    SUBMIT_TO_SLAVE:  output = DoTask(input.data);
    num_idle_slaves--;
  }
  else if (num_idle_slaves < num_slaves) // else insure progress condition
    receive_task_output();               // by blocking until a slave replies
  else break;
} // termination condition:  _after_ all slaves idle, next input was NOTASK
```

The code for `wait_until_an_idle_slave()` can be expanded as follows.

```
void wait_until_an_idle_slave() {
  do {
    while ( result_is_available(&input, &output) ) {
      action = CheckTaskResult(input.data, output.data);
      if (action == UPDATE)
        UpdateSharedData(input.data, output.data);
      if (action == REDO) /* Use updated shared data, when redoing */
        SUBMIT_TO_SLAVE:  output = DoTask(input.data);
      num_idle_slaves++;
    } while (num_idle_slaves == 0);
  }
```

Note that the term *result* refers to an '(input,output)' pair. The routine CheckTaskResult() returns an *action*, which determines the control structure for a parallel algorithm. A common definition is:

```
TOPC_ACTION CheckTaskResult( void *input, void *output ) {
  if (output == NULL) return NO_ACTION;
  else if ( TOPC_is_up_to_date() ) return UPDATE;
  else return return REDO; }
```

TOPC_is_up_to_date() returns true if and only if during the interval between when the task input was originally generated and when the task output was returned by the most recent slave, no other slave process had returned a task output during the interim that had caused the shared data to be modified through an UPDATE action. An UPDATE action causes UpdateSharedData() to be invoked on each process. Further discussion can be found in Section 4.4 [TOP-C Utilities], page 10.

3.2 Three Key Concepts for TOP-C

The 'TOP-C' programmer's model is based on three key concepts:

1. *tasks* in the context of a master/slave architecture;
2. global *shared data* with lazy updates; and
3. *actions* to be taken after each task.

Task descriptions (task inputs) are generated on the master, and assigned to a slave. The slave executes the task and returns the result to the master. The master may update its own private data based on the result, or it may update data on all processes. Such global updates take place on each slave after the slave completes its current task. Updates are *lazy* in that they occur only after a task completes, although it is possible to issue a non-binding request to 'TOP-C' to abort the current tasks (Section 8.2 [Aborting Tasks], page 23). A SPMD (Single Program Multiple Data) style of programming is encouraged.

In both shared and distributed memory architectures, one must worry about the order of reads and writes as multiple slaves autonomously update data. The utilities below are meant to ease that chore, by supporting the ease of the SPMD programming style, while still maintaining good efficiency and generality for a broad range of applications. The software can easily be ported to a variety of architectures.

3.3 Distributed and Shared Memory Models

'TOP-C' provides a single API to support three primary memory models: *distributed memory*, *shared memory* and *sequential memory*. (The last model, sequential memory, refers to a single sequential, non-parallel process.) On a first reading, one should think primarily of the distributed memory model (distributed nodes, each with its own private memory). Most programs written for distributed memory will compile without change for sequential memory. 'TOP-C' is designed so that the same application source code may operate efficiently both under distributed and under shared memory. In order to also compile for shared memory hardware (such as SMP), additional hints to 'TOP-C' may be necessary.

In shared memory architectures, *all* data outside of the four callback functions is shared, by default. Hence, an UPDATE action under shared memory causes only the master process to invoke `UpdateSharedData()`. To avoid inconsistencies in the data, by default 'TOP-C' arranges that no slave process may run `DoTask()` while `UpdateSharedData()` is running. 'TOP-C' also provides support for finer levels of granularity through application-defined private variables and critical sections. Further discussion can be found in Section 8.4 [Shared Memory Model], page 26.

4 Writing 'TOP-C' Applications

This chapter assumes a knowledge of the basic concepts in Chapter 3 [Overview], page 3. In particular, recall Section 3.1.1 [Structure of a TOP-C Program], page 4.

4.1 The Main TOP-C Library Calls

Every 'TOP-C' application must include a '`topc.h`' header, open with `TOPC_init()`, call `TOPC_master_slave()` one or more times, and then close with `TOPC_finalize()`.

`#include <topc.h>`
Required at head of any file using TOPC library calls.

`void` **TOPC_init** (`int *`*argc*, `char ***`*argv*) Function

Required before first occurrence of `TOPC_master_slave()`; Recommended to place this as first executable statement in `main()`. It will strip off extra 'TOP-C' and communication layer arguments such as '`--TOPC-stats`', which are added by 'TOP-C'.

`void` **TOPC_finalize** (`void`) Function

Placed after last 'TOP-C' command.

`void` **TOPC_master_slave** Function

(TOPC_BUF (**generate_task_input*)(),
TOPC_BUF (**do_task*)(void *input),
TOPC_ACTION (**check_task_result*)(void *input, void *output),
void (**update_shared_data*)(void *input, void *output)
)

Primary call, passed four application callbacks to 'TOP-C'. One can have multiple calls to `TOPC_master_slave()`, each invoking different callback functions, between `TOPC_init()` and `TOPC_finalize()`.

A task input or task output is simply a buffer of bytes, specified by `TOPC_MSG()`.

`TOPC_BUF` **TOPC_MSG** (`void *`*buf*, `int` *buf_size*) Function

Must be returned by `GenerateTaskInput()` and `DoTask()`. Specifies arbitrary user data structure. 'TOP-C' will make a copy of *buf* in 'TOP-C' space. It remains the responsibility of the application to free or reuse the space of the original buffer *buf*. If `TOPC_MSG(NULL, 0)` is called, a `NULL` pointer will be received at the destination. (See Section 8.3.2 [Large Buffers and TOPC_MSG_PTR], page 24, for `TOPC_MSG_PTR`, to avoid copying very large buffers, where the overhead is unacceptable.)

EXAMPLE:

```
TOPC_BUF convert_string_to_msg( char *mystring ) {
    if (mystring == NULL) return TOPC_MSG(NULL,0);
    else return TOPC_MSG(mystring, strlen(mystring)+1);
}
```

4.2 Callback Functions for `TOPC_master_slave()`

The application writer must define the following four callback functions (although the last can be `NULL`). The *callback* terminology is based on C concepts. In a more object-oriented style, one would view user callbacks as instantiation of abstract methods in a user-defined subclass. The first two functions return a `TOPC_BUF`, which is produced by `TOPC_MSG()`.

`TOPC_BUF` **GenerateTaskInput** (`void`) Function

executes on master; returns a data structure specified by `TOPC_MSG(`*buf*`,` *buf_size*`)`. It should return `NOTASK`, when there are no more tasks, and it should be prepared to return `NOTASK` again if invoked again.

`TOPC_BUF` **DoTask** (`void *`*input*) Function

executes on slave; operates on the result of `GenerateTaskInput()`; returns a data structure specified by `TOPC_MSG(`*buf*`,` *buf_size*`)`. *buf* must be a static or global user buffer.

`TOPC_ACTION` **CheckTaskResult** (`void *`*input*`,` `void *`*output*) Function

executes on master; operates on the result of `DoTask()`; returns an *ACTION* that determines what happens to the task next. The terminology *result* refers to an '`(input, output)`' pair. An easy way to write `CheckTaskResult()` appears in the example for the utility `TOPC_is_up_to_date()`. When returning the action `UPDATE`, it works to first modify the input and output buffers. `UpdateSharedData()` will then be invoked with the modified buffers. See Section 4.4 [TOP-C Utilities], page 10, for more details.

`void` **UpdateSharedData** (`void *`*input*`,` `void *`*output*) Function

executes on master and all slaves; operates on the result of `DoTask()`, and the original task returned by `GenerateTaskInput()`; called only if `CheckTaskResult()` returned `UPDATE`; useful for updating global variables in all processes; The pointer argument, *update_shared_data*, of `TOPC_master_slave()` may be `NULL` if an application never requests an `UPDATE` action. In a shared memory environment, only the master calls `UpdateSharedData()`. See Section 8.4 [Shared Memory Model], page 26, for more details.

Note that in defining the above callback functions, C allows one to replace the `(void *)` declaration of the arguments by specific pointer types. Note that the buffers of any message parameters (*input* and *output*) of `DoTask()` or `CheckTaskResult()` are part of TOP-C space. Such buffers may be freed by TOP-C on exit from the callback function. An application wishing to use the buffer after the callback exits must explicitly save a copy into the application's own space.

4.3 Actions Returned by `CheckTaskResult()`

A *TOP-C result* is an (*input*, *output*) pair corresponding to an invocation of `DoTask()`. 'TOP-C' passes the result to `CheckTaskResult()`. The return value allows the application to tell 'TOP-C' what further actions to take. The actions returned by `CheckTaskResult()` are:

`TOPC_ACTION` **NO_ACTION** Action

C constant, causing no further action for task

`TOPC_ACTION` **UPDATE** Action

C constant, invoking `UpdateSharedData( void *`*input*`, void *`*output*) (see below) also updates bookkeeping for sake of `TOPC_is_up_to_date()` (see Section 4.4 [TOP-C Utilities], page 10)

`TOPC_ACTION` **REDO** Action

Invoke `DoTask()` on original task input again, and on the same slave that previously executed the task; useful if shared data has changed since original invocation of `DoTask()` (see `TOPC_is_up_to_date()`, below). See Section 7.2 [Strategies for Greater Concurrency], page 21, for slave strategies to efficiently process a `REDO` action.

`TOPC_ACTION` **CONTINUATION (void ****next_input* **)** Action

`CONTINUATION(`*next_input*`)` is a parametrized action that may be returned by `CheckTaskResult()`, after which `DoTask(` *next_input* `)` is called on the original slave. This is useful if only the master can decide whether task is complete, or if the master wishes to supply additional input messages needed for the task. Note that `REDO` is essentially equivalent to `CONTINUATION(` *original_task_input* `)`. Note that any pending calls to `UpdateSharedData()` will have occurred on the slave before the new call to `DoTask()`. Hence, this allows an extended conversation between master and slave, in which the slave continues to receive updates of the shared data before each new input from the master. Note also that even though a `CONTINUATION` action returns to the original slave, any previous pointers to input buffers on that slave (and pointers to output buffers from intervening `UPDATE` actions) will no longer be valid. Useful data from previous buffers should have been copied into global variables on the slave. In the case of the shared memory model, those global variables must be thread-private. (see Section 8.4.2 [Thread-Private Global Variables], page 28)

It is possible for `CheckTaskResult(`*input*`, `*output*`)` to modify the buffer data in its two arguments, *input* and *output*, in which case the modifications will be visible to any further callback functions processing the current task. This practice makes the code more difficult to maintain, and is not recommended when other solutions are available.

4.4 TOP-C Utilities

'TOP-C' also defines some utilities.

TOPC_BOOL **TOPC_is_up_to_date** (void) Function

returns TRUE or FALSE (1 or 0); returns TRUE if and only if `CheckTaskResult()` has not returned the result `UPDATE` (invoking `UpdateSharedData()`) between the time when `GenerateTaskInput()` was originally called on the current task, and the time when the corresponding `CheckTaskResult()` was called. Typical usage:

```
TOPC_ACTION CheckTaskResult( void *input, void *output )
{ if (input == NULL) return NO_ACTION;
  else if (TOPC_is_up_to_date()) return UPDATE;
  else return REDO;
}
```

int **TOPC_rank** (void) Function

Unique ID of process or thread. Master always has rank 0. Slaves have contiguous ranks, beginning at 1.

TOPC_BOOL **TOPC_is_master** (void) Function

Returns boolean, 0 or 1, depending on if this is master. Equivalent to `TOPC_rank() == 0`.

int **TOPC_num_slaves** (void) Function

Total number of slaves.

int **TOPC_num_idle_slaves** (void) Function

Total number of idle slaves (not doing task, update or other action).

int **TOPC_node_count** (void) Function

Total number of processes or threads. Equivalent to `TOPC_num_slaves() + 1`.

TOPC_BOOL **TOPC_is_REDO** (void) Function
TOPC_BOOL **TOPC_is_CONTINUATION** (void) Function
void **TOPC_abort_tasks** (void) Function
TOPC_BOOL **TOPC_is_abort_pending** (void) Function

See Chapter 8 [Advanced Features], page 23 for descriptions.

5 Compiling and Invoking 'TOP-C' Applications

A 'TOP-C' application can be compiled once, and then linked to your choice of a run-time library for either a sequential, distributed memory or shared memory architecture. The two shell scripts '`bin/topcc`' and '`bin/topc++`' are used instead of '`gcc`' and '`g++`' (or other C/C++ compilers).

5.1 Compiling TOP-C Applications

The TOP-C application file must contain

```
#include <topc.h>
```

It must make calls to

```
TOPC_init(...);
TOPC_master_slave(...);
TOPC_finalize();
```

as describe in Section 3.1.1 [Structure of a TOP-C Program], page 4. The application file is compiled by one of:

`topcc --seq` *myfile.c*
`topcc --mpi` *myfile.c*
`topcc --pthread` *myfile.c*

according to whether the target computer architecture will be sequential ('`--seq`': single processor), distributed memory ('`--mpi`': networked CPU's), or shared memory ('`--pthread`': SMP or other shared memory architecture with a POSIX threads interface). `topcc` is a substitute for `cc` or `gcc`, and creates an '`a.out`' file. (Similarly, `topc++` exists as a substitute for `c++` or `g++`.) There are man files,

'`doc/topcc.1`', '`doc,topc++.1`'

with further information on `topcc` and `topc++`. If installed, `man topcc` and `man topc++` exist.

The same object file may be relinked to use different '`TOP-C`' memory models without recompiling the object file.

`topcc -c` *myapp.c*
`topcc --seq -o` *myapp-seq myapp.o*
`topcc --mpi -o` *myapp-mpi myapp.o*

For large applications, it may be preferable to directly invoke the '`TOP-C`' libraries and include files. For such situations, `topc-config` exists. The following is exactly equivalent to `topcc --mpi myfile.c` (assuming you configured '`TOP-C`' using `gcc`).

```
gcc `topc-config --cflags` --mpi myfile.c `topc-config --libs`
```

Type `topc-config` with no arguments for a full set of command line options.

For the rest of this chapter, we standardize our description for `topcc`. However, `topc++` is equally valid wherever `topcc` is mentioned.

5.2 Command Line Options in TOP-C Applications

TOP-C searches for TOP-C parameters in the following locations, in order:

1. values of variables `TOPC_OPT_xxx` in the application code before `TOPC_init()`
2. the file '`.topcrc`' in the home directory
3. the environment variable `TOPC_OPTS`
4. arguments on the command line in the form `--TOPC-xxx`

The file '`.topcrc`' and the environment variable `TOPC_OPTS` specify parameters in the same format as on the command line. Later assignments of an option override earlier assignments.

For a brief synopsis of application command line options, type:

```
topcc myapp.c
./a.out --TOPC-help
[ OR FOR MORE INFORMATION:  ./a.out --TOPC-help --TOPC-verbose  ]
```

Currently, this will display the following.

```
Usage:  ./a.out [TOPC_OPTION ...] [OTHER_OPTIONS ...]

  where TOPC_OPTION is one of:
--TOPC-help[=<0/1>]        display this message   [default: false]
--TOPC-stats[=<0/1>]       print stats before & after [default: false]
--TOPC-verbose[=<0/1>]     set verbose mode       [default: false]
--TOPC-num-slaves=<int>    number of slaves (sys-defined default)
                                                  [default: -1]
--TOPC-aggregated-tasks=<int> number of tasks to aggregate
                                                  [default: 1]
--TOPC-slave-
wait=<int>    secs before slave starts (use w/ gdb attach)
                                                  [default: 0]
--TOPC-slave-timeout=<int> dist mem: secs to die if no msgs, 0=never
                                                  [default: 1800]
--TOPC-
trace=<int>         trace (0: notrace, 1: trace, 2: user trace fncs.)
                                                  [default: 2]
--TOPC-procgroup=<string>  procgroup file (--mpi)
                                                  [de-
fault: "./procgroup"]
--TOPC-topc-log=<string>   NOT IMPL: log file for TOPC output ("-
" = stdout)
                                                  [default: "-"]
--TOPC-safety=<int>        [0..20]: higher turns off optimizations,
                         (try with --TOPC-verbose) [default: 0]
```

For each option, '--TOPC-*PARAM*', there is a corresponding C/C++ variable, TOPC_OPT_*PARAM*. This variable is of type int or (char *). If the application program sets the value before a call to TOPC_init(), these variables will act as defaults. For example, to turn off tracing by default, write:

```
int main( int argc, char *argv[] ) {
  TOPC_OPT_trace = 0;
  TOPC_init( &argc, &argv );
  ...
}
```

The option '--TOPC-trace' causes the task input and task output to be traced and printed as they are passed across the network. The action of a task is also printed. If an application is called with '--TOPC-trace=2' (default) and if the variables TOPC_OPT_trace_input and TOPC_OPT_trace_result are set to pointers to functions then those functions are called, and they may print additional information. TOPC_OPT_trace_input must be set to a function of one variable: void * *input*. TOPC_OPT_trace_result must be set to a function of two variables: void * *input*, void * *output*. When using C++, the function pointers must be cast to `TOPC_trace_input_ptr` or `TOPC_trace_result_ptr` before being assigned to `TOPC_OPT_trace_input` or `TOPC_OPT_trace_result`, respectively. For an example, see '`examples/parfactor.c`' in the TOP-C distribution.

The option '`--TOPC-stats`' prints statistics (running times, etc.) and information about the conditions of an invocation of a TOP-C application before and after a run. The option '`--TOPC-verbose`' (set by default) displays TOP-C warnings. With –TOPC-help, it provides additional information.

For the usage of '`--TOPC-procgroup`', see Section 5.4 [Invoking a TOP-C Application in Distributed Memory], page 14. That section also explains on what hosts the slaves run when '`--TOPC-num-slaves`' indicates a different number of slaves than the '`procgroup`' file.

For the usage of '`--TOPC-aggregated-tasks`', see Section 7.3 [Improving Performance], page 21. For the usage of '`--TOPC-slave-wait`', see Section 6.5 [Stepping Through a Slave], page 19. For the usage of '`--TOPC-safety`', see Section 5.2 [Command Line Options], page 12. For the usage of the other options, see the [Concept Index], page 41.

'`TOP-C`' recognizes `--` as terminating all option processing, according to standard UNIX conventions.

5.3 Invoking a TOP-C Application in Sequential Memory

For example,

```
topcc --seq -g -O0 myfile.c
```

compiles a sequential version for debugging using `gdb` (see section "Summary" in *The GNU debugger*), for example. This is usually a first step in debugging a TOP-C application, since sequential debugging is easier than parallel debugging.

5.4 Invoking a TOP-C Application in Distributed Memory

Linking using the '`--mpi`' option (default) allows an application to execute using a distributed memory model of networked processors. The '`TOP-C`' distribution includes a subset MPI[1] implementation '`MPINU`', sufficient to run '`TOP-C`' applications.

```
topcc --mpi myapp.c
./a.out
```

The application must then create the remote slave processes at runtime. If you use '`MPINU`' (the default configuration of '`TOP-C`', then the remote slave processes are specified by a '`procgroup`' file. Otherwise, the startup mechanism depends on your '`MPI`' implementation.

5.4.1 Writing Procgroup Files for '`MPINU`'

'`MPINU`' is built into the default configuration of the '`TOP-C`' library and uses the '`procgroup`' mechanism to specify slave processes. (If you prefer to use a different '`MPI`' dialect, '`TOP-C`' will work, but '`src/Makefile.in`' must be modified, and that '`MPI`' dialect may use a different mechanism for introducing slave processes.)

[1] MPI is *Message Passing Interface*, see `http://www.mpi-forum.org/`

When the application binary is executed under the default, it looks at the current directory for a file,

'`procgroup`'

The *procgroup file* determines the number and location of the slave processes. The *procgroup file* need only be visible from the master process. If one prefers, one can specify an alternate *procgroup file* via the syntax as in the following example:

```
./a.out --TOPC-procgroup=../myprocgroup.big
```

The '`TOP-C`' distribution includes a file '`bin/procgroup`' as an example of the procgroup format. The file must contain a line:

```
local 0
```

for the master process. It must also contain a line for each slave process in one of the following forms:

```
hostname 1 full_pathname
hostname 1 -
hostname 1 ./relative_pathname
hostname 1 ../relative_pathname
```

where `hostname` is the remote host. The pathname `-` means to use the same pathname for the slave on the remote host as for the master on the current host. A relative pathname, such as `./a.out` or `../new_arch/a.out`, specifies a pathname relative to the pathname of the binary of the master on the current host.

Most of the time, it is simplest to just include a full pathname or else `-` in the '`procgroup`' file. The relative pathnames are useful for a shared filesystem with binaries compiled for different architectures. For example, the procgroup file might include relative paths '`../sparc/a.out`', '`../alpha/a.out`' and '`../linux/a.out`'. If you invoke '`full_path/sparc/a.out`', this will yield a master running '`full_path/sparc/a.out`' and three slaves running each of the three architectures.

The full principles are as follows. Let SLAVE_PATH be the path of the slave as given in the procgroup file, and let MASTER_DIR be the directory of the master process as invoked on on the command line.

```
SLAVE_PATH absolute:
  slave binary image is SLAVE_PATH
SLAVE_PATH relative and MASTER_DIR absolute:
  slave binary image is MASTER_DIR/SLAVE_PATH
SLAVE_PATH relative and MASTER_DIR relative:
  slave binary image is $PWD/MASTER_DIR/SLAVE_PATH
SLAVE_PATH is - and master process invoked on command line as MASTER_BIN:
  slave binary image is MASTER_BIN  (if MASTER_BIN is absolute path)
  or $PWD/MASTER_BIN (if MASTER_BIN is relative path)
If the procgroup line contains command line arguments,
  those command line arguments are passed to the slave application
  as its first arguments, and any arguments on the master command
  line are appended to the list of arguments.
```

TOP-C assumes that your application does not change the working directory before calling `TOPC_init()`.

By default, 'TOP-C' uses the procgroup file in the current directory. You can choose an explicit procgroup file via a program variable, `TOPC_OPT_procgroup="/project/myprocgroup";`, or via a command-line option, `/project/sparc/app --TOPC-procgroup=/project/myprocgroup`. See Section 5.2 [Command Line Options], page 12.

If the command line option '`--TOPC-num-slaves=`*int*' is given, and if *int* is less than the number of slaves in the '`procgroup`' file, then 'TOP-C' will use the first *int* slaves from the file. If *int* is more than the number of slaves in the '`procgroup`' file, then 'TOP-C' will use all of the given slaves, and then create additional processes on the remote hosts, by returning to the beginning of the '`procgroup`' file and re-reading the list of slave host/binaries until *int* slaves have been created in all.

It is recommended to use only `localhost` during initial development.

The environment variable, `SSH` (default value `ssh`) is used to invoke the remote host. If, for example, your site uses `rsh` instead of `ssh`, the following code, when executed before `TOPC_init()` will produce this effect.

```
putenv("SSH=rsh");
```

Alternatively, type `SSH=rsh` (sh/bash, etc.) or `setenv SSH rsh` (csh/tcsh, etc.) in the UNIX shell before invoking the TOP-C application.

5.4.2 If Slaves Fail to Start

If some slave processes start, but not others, then try executing the following simple program.

```
#include "topc.h"
int main(int argc, char *argv) {
  char host[100];
  printf("%s connecting ...\n", gethostname(host, 99));
  TOPC_init(&argc, &argv);
  printf("... %s connected.\n", gethostname(host, 99));
  TOPC_finalize();
}
```

If the slave processes fail to start up or fail to respond to the master and if you are using 'MPINU' (default configuration of 'TOP-C', one other debugging resource is available. If an application fails to start up, then 'TOP-C' leaves in the '`/tmp`' directory a file

'`/tmp/mpinu-ssh.$$`'

where `$$` is the process id. The file shows the commands that '`TOP-C/mpinu`' tried to use to start up the slave process. By examining and even manually executing those commands from the terminal, one can often deduce the difficulty in creating the slave processes.

Test the ability of your computer facility to execute remote processes *without passwords* by typing: `ssh REMOTE_HOST pwd`. If the problem is that `ssh` is not working, try setting the environment variable `SSH` to `ssh` or other site-specific setting. See Section 5.4.1 [Procgroup Files], page 14.

If you are using `ssh` (default if the environment variable `SSH` is not set), and if it requires a password then the following commands within UNIX may allow ssh to operate on your local cluster without passwords.

```
ssh-keygen -t dsa       [accept default values]
ssh-keygen -t rsa       [accept default values]
cat ~/.ssh/id*.pub >> ~/.ssh/authorized_keys
```

For security, be sure that ~/.ssh has no read permission for other users.

5.5 Invoking a TOP-C Application in Shared Memory

Linking using the '`--pthread`' option allows an application to execute using POSIX threads. Note that the '`TOP-C`' memory model for shared memory has some small variations from the distributed memory model. The largest potential source of incompatibility is if your `DoTask()` routine modifies any global or static variables. If so, such variables will need to be declared *thread private*. Check your source code to see if this affects you.

Second, if you encounter insufficient performance, you may consider experimenting with *fine grain thread parallelism*. The default '`TOP-C`' algorithm for shared memory allows `UpdateSharedData()` to begin executing only after each current invocation of `DoTask()` completes. This can be modified by an application for finer grain parallelism. See Section 8.4 [Optimizing TOP-C Code for the Shared Memory Model], page 26, for details in either of the above cases.

Note also that while a '`TOP-C`' application object file can usually be linked using an arbitrary '`TOP-C`' memory model without recompiling, there are some circumstances where you may first need to recompile the application source using `topcc --pthread`.

6 Debugging and Tracing

If the difficulty is that the application fails to start in the distributed memory model (using `topcc --mpi`), then read Section 5.4.2 [Slaves Fail to Start], page 16, for some debugging techniques. Note also that TOP-C ignores SIGPIPE. This is because TOP-C employs the SO_KEEPALIVE option, and the master process would otherwise die if a slave process were to die. SO_KEEPALIVE is needed for robustness when slave processes execute long tasks without communicating with the master process. The rest of this section assumes that the application starts up correctly.

6.1 Debugging by Limiting the Parallelism

First, compile and link your code using `topcc --seq --TOPC-safety=20 -g -O0`, and make sure that your application works correctly sequentially. Only after you have confidence in the correctness of the sequential code, should you begin to debug the parallel version.

If the application works correctly in sequential mode, one should debug in the context of a single slave. It is convenient to declare the remote slave to be `localhost` in the '`procgroup`' file, in order to minimize network delays and so as not to disturb users of other machines. In this case, the code is "almost" sequential. An easy way to do this is:

```
topcc --mpi --TOPC-num-slaves=1 -g -O0
```

Next, one should test on two slaves, and finally all possible slaves.

6.2 Debugging with '--TOPC-safety'

The command-line option '`--TOPC-safety=`*val*' provides assistance for debugging parallel programs. At higher values of *val*, optimizations that do not change the correctness of the program are converted to safer equivalents. A good strategy is to test if '`--TOPC-safety=20`' causes the bug to go away, and if so, progressively lower *val* toward zero, until the bug reappears. The value at which the bug reappears indicates what '`TOP-C`' optimization feature is not being used correctly. If the bug still exists at '`--TOPC-safety=20`', one should next try compiling with the '`--seq`' flag and use a debugger to debug the sequential code.

The effects due to different safety levels are subject to change. To see the current effects, invoke any TOP-C application as follows

```
./a.out --TOPC-help --TOPC-verbose
```

and you will see something like:

```
safety: >=0: all; >=2: no TOP-C memory mgr (uses malloc/free);
>=4: no TOPC_MSG_PTR; >=8: no aggreg.;
>=12: no TOPC_abort_tasks; >=14: no receive thread on slave;
>=16: default atomic read/write for DoTask, UpdateSharedData;
 =19: only 2 slaves; >=20: only 1 slave
(AGGREGATION NOT YET IMPLEMENTED)
```

Values higher than 4 cause `TOPC_MSG_PTR()` to act as if `TOPC_MSG()` was called instead. Values higher than 12 cause `TOPC_abort_tasks()` to have no effect. Values higher than 14 imply that a single thread in the slave process must receive messages and execute `DoTask()`. Normally, '`TOP-C`' arranges to overlap communication and computation on the slave by setting up a separate thread to receive and store messages from the master. Values higher than 15 imply that '`TOP-C`' will use malloc instead of trying to do its own memory allocation (which is optimized for '`TOP-C`' memory patterns). Values higher than 16 imply that all of `DoTask` acts as if a read lock was placed around it, and all of `UpdateSharedData` has a write lock placed aound it. (This has an effect only in the shared memory model where calls to `TOPC_ATOMIC_READ/WRITE` are ignored.) At values of 19 and 20, the number of slaves is reduced to 2 and to 1, regardless of the setting of '`--TOPC-num-slaves`' and the specification in a '`procgroup`' file.

6.3 TOP-C and POSIX signals

If an application handles its own signals, this can create a clash with the TOP-C. In the distributed memory model (–mpi), '`TOP-C`' will create its own signal handlers for `SIGALRM`. This is used in conjunction with `alarm()` to eventually kill runaway slave processes. In addition, if using '`MPINU`', the built-in MPI subset, '`TOP-C`' will create its own handler for `SIGPIPE`. This is in order to allow the master process to detect dead sockets, indicating dead slaves. Finally, for short periods, '`MPINU`' will disable the use of `SIGINT` around calls to `select()`. Nevertheless, if a SIGINT is sent during this period, TOP-C will pass the signal on to the original SIGINT handler of the application.

'`TOP-C`' does not modify signal handlers in the sequential (–seq) or shared memory (–pthread) models. Furthermore, if a different MPI (other than MPINU) is used with TOP-C, TOP-C will only handle `SIGALRM`. However, the other MPI may handle signals itself. See Appendix C [Using a Different MPI with TOP-C], page 39.

6.4 Tracing Messages

If a bug appears as one moves to greater parallelism, one should trace messages between master and slaves (for any number of slaves). This is the default, and it can be enabled on the command line with:

```
./a.out --TOPC-trace=2 args
```

The variable `TOPC_OPT_trace` can be set in the code to dynamically turn tracing on (1 or 2) and off (0) during a single run. A trace value of 2 causes '`TOP-C`' to invoke the application-defined trace functions pointed to by `TOPC_OPT_trace_input/result`. If the application has not defined trace functions, or if `TOPC_OPT_trace` is 1, then the '`TOP-C`' default trace functions are invoked. All message traces are displayed by the master at the time that the master sends or receives the corresponding message.

`void (*)(void *input)` **TOPC_OPT_trace_input** — Variable
`void (*)(void *input, void *output)` **TOPC_OPT_trace_result** — Variable

Global pointer (default is `NULL`) to function returning `void`. User can set it to his or her own trace function to print out data-specific tracing information in addition to generic message tracing of `TOPC_trace`.

```
EXAMPLE:  if you pass integers via TOPC_MSG(), define
  TOPC_trace_input() as:
         void mytrace_input( int *input ) {
           printf("%d",*input);
         }
         TOPC_OPT_trace_input = mytrace_input;
```

Note that the term '`result`' in `TOPC_OPT_trace_result` refers to an '`(input, output)`' pair.

6.5 Stepping Through a Slave Process with '`gdb`'

If you find the master hanging, waiting for a slave message, then the probable cause is that `DoTask()` is doing something bad (hanging, infinite loop, bus/segmentation error, etc.). First try to isolate the bug using a symbolic debugger (e.g. '`gdb`') and the sequential memory model. If your intended application is the shared memory model, you can also use '`gdb`' to set a breakpoint in your '`DoTask`' routine or at the '`TOP-C`' invocation, `do_task_wrapper`.

If the bug only appears in the distributed memory model, you can still symbolically debug `DoTask()` using '`gdb`' (the GNU C debugger) and its `attach` command (see section "Attach" in *The GNU debugger*), which allows you to attach and debug a separate running process. This lets you debug a running slave, if it is running o the same processor. For this strategy, you will want the slave to delay executing to give you time to execute gdb and attach on the remote host or remote thread. The command line option '`--TOPC-slave-wait=30`' will force the slave to wait 30 seconds before processing.

In applying this debugging strategy to an application '`./a.out`', one might see:

```
[ Execute ./a.out in one window for master process ]
gdb ./a.out
(gdb) run --TOPC-trace=1 --TOPC-safety=19 --TOPC-slave-wait=30 args

[ In a second window for a slave process on a different host, now type: ]
ps
  ...
  1492  p4 S    0:00 a.out args localhost 6262 -p4amslave
gdb a.out
...
(gdb) break do_task_wrapper
  Breakpoint 1 at 0x80492ab: file ...
  [ 'break slave_loop' is also useful.  This calls do_task_wrapper ]
(gdb) attach 1492
  Attaching to program 'a.out', process 1492
  0x40075d88 in sigsuspend ()
[ After 30 sec's, traced messages in master window appear, ]
[ for slave, type: ]
(gdb) continue
  Continuing.
  Breakpoint 1, DoTask (input=0x805dc50) at ...

[ Continue stepping through master and slave processes in 2 windows ]
```

If you try to *attach* to a second slave process after attaching to a first slave process, '`gdb`' will offer to kill your first slave process. To avoid this situation, remember to execute `detach` before attaching a second slave process.

6.6 Segmentation faults and other memory problems

Memory bugs are among the most difficult to debug. If you suspect such a bug (perhaps because you are using `TOPC_MSG_PTR`), and you fail to free previously malloc'ed memory, that is a memory leak. If you access a buffer after freeing it, this may cause a segmentation error at a later stage in the program.

If you suspect such a bug (and maybe you should if nothing else worked), it is helpful to use a *malloc or memory debugger*. An excellent recent memory debugger is '`valgrind`'[2]. '`valgrind`' can be directly applied to an application binary, without recompilation or relinking.

An older debugger is '`efence`',[3] `topcc` provides direct support for '`efence`'. '`TOP-C`' will link with the efence library if `--efence` is passed to `topcc` or `topc++`.

```
topcc --efence ...
```

This causes all calls to `malloc` and `free` to be intercepted by the '`efence`' version. Modify the line `LIBMALLOC=` in `topcc` or `topc++` if you use a different library.

---

[2] *valgrind* is available at `http://www.valgrind.kde.org`.

[3] *efence* is available at `http://sources.isc.org/devel/memleak/efence`.

7 Performance and Long Jobs

7.1 Dropping Slow or Dead Slaves

When TOP-C recognizes a dead slave the master terminates communication with that slave, and resubmits the task of that slave to a different slave. (Currently, as of TOP-C 2.5.0, if a slave dies near the end of a computation and after all tasks have been generated, TOP-C may fail to recognize that slave.)

It is sometimes unclear whether a slave process is dead or slower than others. Even a slow slave process may hurt overall performance by causing delays for other processes. TOP-C internally declares a slave process to be "slow" if there are N slaves, and if 3*N other tasks return after the given slave task is "due". If a slow slave has not returned by *slave-timeout* seconds (see Section 5.2 [Command Line Options], page 12), then the slave is considered dead. The master process sends no further tasks to that slave, and sends a replicate of the original task to a new slave.

7.2 Strategies for Greater Concurrency

Strategy 1: SEMI-INDEPENDENT TASKS:
: Define tasks so that most task outputs do not require any update. This is always the case for trivial parallelism (when tasks are independent of each other). It is also often the case for many examples of search and enumeration problems.

Strategy 2: CACHE PARTIAL RESULTS:
: Inside `DoTask()` and `UpdateSharedData()`, save partial computations in global private variables. Then, in the event of a `REDO` action, '`TOP-C`' guarantees to invoke `DoTask()` again on the original slave process or slave thread. That slave may then use previously computed partial results in order to shorten the required computation. Note that pointers on the slave to input and output buffers from previous `UPDATE` actions and from the original task input will no longer be valid. The slave process must copy any data it wishes to cache to global variables. In the case of the shared memory model, those global variables must be thread-private. (see Section 8.4.2 [Thread-Private Global Variables], page 28) Note the existence of `TOPC_is_REDO()` for testing for a `REDO` action.

Strategy 3: MERGE TASK OUTPUTS:
: Inside `CheckTaskResult()`, the master may merge two or more task outputs in an application independent way. This may avoid the need for a `REDO` action, or it may reduce the number of required `UPDATE` actions.

7.3 Improving Performance

If your application runs too slowly due to excessive time for communication, consider running multiple slave processes on a single processor. This allows one process to continue computing while another is communicating or idle waiting for a new task to be generated by the master.

If communication overhead or idle time is still too high, consider if it is possible to increase the granularity of your task. TOP-C can aggregating several consecutive tasks as a single larger task to be performed by a single process. This amortizes the network latency of a single network message over several tasks. For example, you can do combine 5 tasks by invoking '--TOPC-aggregated-tasks=5' on the command line of the application. Alternatively, execute the statement:

```
TOPC_OPT_aggregated_tasks=5;
```

before `TOPC_master_slave()`. In this situation, the five task outputs will also be bundled as a single network message. Currently (TOP-C 2.5.0), this works only if all tasks return `NO_ACTION`. TOP-C will signal an error if `TOPC_OPT_aggregated_tasks > 1` and any action other than `NO_ACTION` is returned.

Other useful techniques that may improve performance of certain applications are:

1. set up multiple slaves on each processor (if slave processors are sometimes idle)
2. re-write the code to bundle a set of tasks as a single task (to improve the granularity of your parallelism)

PERFORMANCE ISSUE FOR MPI:

If you have a more efficient version of 'MPI' (perhaps a vendor version tuned to your hardware), consider replacing `LIBMPI` in '.../top-c/Makefile' by your vendor's 'limbpi.a' or 'libmpi.so', and delete or modify the the `LIBMPI` target in the 'Makefile'. Alternatively, see the appendix, Appendix C [Using a Different MPI with TOP-C], page 39, for a more general way to use a different MPI dialect.

PERFORMANCE ISSUE FOR SMP (POSIX threads):

Finally under 'SMP', there is an important performance issue concerning the interaction of 'TOP-C' with the operating system. First, the vendor-supplied compiler, `cc`, is recommended over `gcc` for 'SMP', due to specialized vendor-specific architectural issues. Second, if a thread completes its work before using its full scheduling quantum, the operating system may yield the CPU of that thread to another thread — potentially including a thread belonging to a different process. There are several ways to defend against this. One defense is to insure that the time for a single task is significantly longer than one quantum. Another defense is to ask the operating system to give you at least as many "run slots" as you have threads (slaves plus master). Some operating systems use `pthread_setconcurrency()` to allow an application to declare this information, and 'TOP-C' invokes `pthread_setconcurrency()` where it is available. However, other operating systems may have alternative ways of tuning the scheduling of threads, and it is worthwhile to read the relevant manuals of your operating system.

7.4 Long Jobs and Courtesy to Others

In the distributed memory model, infinite loops and broken socket connections tend to leave orphaned processes running. In the 'TOP-C' distributed memory model, a slave times out if a task lasts longer than a half hour or if the master does not reply in a half hour. This is implemented with the UNIX system call, `alarm()`.

A half hour (1800 seconds) is the default timeout period. The command line option `--TOPC-slave-timeout=`*num* allows one to change this default. If *num* is 0, then there is no timeout and 'TOP-C' makes no calls to `SIGALRM`.

The application writer may also find some of the following UNIX system calls useful for allowing large jobs to coexist with other applications.

`setpriority(PRIO_PROCESS,getpid(),`*prio*`)`
: `#include <unistd.h>`
`#include <sys/resource.h>`
— *prio* = 10 still gives you some CPU time. *prio* = 19 means that any job of higher priority always runs before you. Place in `main()`.

`setrlimit(RLIMIT_RSS, &`*rlp*`)`
: `#include <sys/resource.h>`
`struct rlimit rlp;`
`rlp.rlim_max = rlp.rlim_cur = `*SIZE*`;`
— *SIZE* is RAM limit (bytes). If your system has significant paging, the system will prefer to keep your process from growing beyond SIZE bytes of resident RAM. Even if you set nice to priority 20, this is still important. Otherwise you may cause someone to page out much of his or her job in your favor during one of your infrequent quantum slices of CPU time. Place in `main()`. (Not all operating systems enforce this request.)

8 Advanced Features of 'TOP-C'

It is best to postpone reading this section until the basic features discussed in the previous chapters are clear.

8.1 Testing for Task Continuations and Redos

`TOPC_BOOL` **TOPC_is_REDO** (`void`) Function

`TOPC_BOOL` **TOPC_is_CONTINUATION** (`void`) Function

These return 0 (false) or 1 (true), according to whether the current call to `DoTask()` was a result of a `REDO` or `CONTINUATION()` action, respectively. The result is is not meaningful if called outside of `DoTask()`.

8.2 Aborting Tasks

`void` **TOPC_abort_tasks** (`void`) Function

`TOPC_BOOL` **TOPC_is_abort_pending** (`void`) Function

`TOPC_abort_tasks()` should be called in `CheckTaskResult()`. 'TOP-C' then makes a best effort (no guarantee) to notify each slave. TOP-C does not directly abort tasks. However, `TOPC_is_abort_pending()` returns 1 (true) when invoked in `DoTask()` on a slave. A typical `DoTask()` callback uses this to poll for an abort request from the master, upon which it returns early with a special task output. At the beginning of the next new task, `REDO` or `CONTINUATION`, 'TOP-C' resets the pending abort to 0 (false). See '`examples/README`' of the 'TOP-C' distribution for example code.

8.3 Memory Allocation for Task Buffers

The principle of memory allocation in 'TOP-C' is that if an application allocates memory, then it is the responsibility of the application to free that memory. This issue typically arises around the issue of task buffers (see Section 3.1.3 [Task Buffers], page 5) and calls to `TOPC_MSG(`*buf*`,`*buf_size*`)`. An application often calls *buf* `= malloc(...);` or *buf* `= new ...;` (in C++) and copies data into that buffer before the call to `TOPC_MSG`. Since the last action of `GenerateTaskInput()` or `DoTask()` is typically to return `TOPC_MSG(`*buf*`,`*buf_size*`)`, there remains the question of how to free *buf*.

8.3.1 Avoiding `malloc` and `new` with Task Buffers

The best memory allocation solution for task buffers is to implement the buffers as local variables, and therefore on the stack. This avoids the need for `malloc` and `new`, and the question of how to later free that memory. If you use `TOPC_MSG` (as opposed to `TOPC_MSG_PTR`, see Section 8.3.2 [Large Buffers and TOPC_MSG_PTR], page 24), then recall that `TOPC_MSG` copies its buffer to a separate `TOP-C` space. For example,

```
{ int x;
   ...
  return TOPC_MSG(&x, size_of(x));
}
```

If your task buffer is of fixed size, one can allocate it as a character array on the stack: `char buf[BUF_SIZE];`. If your buffer contains variable size data, consider using `alloca` in place of `malloc` to allocate on the stack.

```
{ ...
  buf = alloca(buf_size);
  return TOPC_MSG(buf, buf_size);
}
```

In all of the above cases, there is no need to free the buffer, since `TOPC_MSG` will make a 'TOP-C'-private copy and the stack-allocated buffer will disappear when the current routine exits. Note that `alloca` may be unavailable on your system. Alternatively, the use of `alloca` may be undesirable due to very large buffers and O/S limits on stack size. In such cases, consider the following alternative.

```
{ TOPC_BUF tmp;
  ...
  buf = malloc(buf_size);
  tmp = TOPC_MSG(buf, buf_size);
  free(buf);
  return tmp;
}
```

8.3.2 Using `TOPC_MSG_PTR()` to Avoid Copying Large Buffers

If the cost of copying a large buffer is a concern, 'TOP-C' provides an alternative function, which avoids copying into 'TOP-C' space.

TOPC_BUF **TOPC_MSG_PTR** (void **buf*, int *buf_size*) Function

Same as TOPC_MSG(), except that it does not copy *buf* into 'TOP-C' space. It is the responsibility of the application not to free or modify *buf* as long as 'TOP-C' might potentially pass it to an application callback function.

TOPC_MSG_PTR() is inherently dangerous, if the application modifies or frees a buffer and 'TOP-C' later passes that buffer to a callback function. It may be useful when the cost of copying large buffers is an issue, or if one is concerned about 'TOP-C' making a call to malloc(). Note that the invocation

```
./a.out --TOPC-safety=4
```

automatically converts all calls to TOPC_MSG_PTR() into calls to TOPC_MSG(). This is useful in deciding if a bug is related to the use of TOPC_MSG_PTR().

An application should not pass a buffer on the stack to TOPC_MSG_PTR(). This can be avoided either by declaring a local variable to be 'static', or else using a global variable (or a class member in the case of C++). In such cases, it is the responsibility of the application to dynamically create and free buffers. An example of how this can be done follows in the next section.

Note that if the application code must also be compatible with the shared memory model, then the static local variable or global variable must also be *thread-private* (Section 8.4.2 [Thread-Private Global Variables], page 28).

For examples of coding with TOPC_MSG_PTR() that are compatible with all memory models, including the shared memory model, see 'examples/README' and the corresponding examples in the 'TOP-C' distribution.

8.3.3 Allocation and Freeing of Task Buffers for TOPC_MSG_PTR()

Recall the syntax for creating a message buffer of type TOPC_BUF using TOPC_MSG_PTR(*buf*, *buf_size*). The two callback functions GenerateTaskInput() and DoTask() both return such a message buffer. In the case of GenerateTaskInput(), 'TOP-C' saves a copy of the buffer, which becomes an input argument to CheckTaskResult() and to UpdateSharedData on the master. Hence, if *buf* points to a temporarily allocated buffer, it is the responsibility of the 'TOP-C' callback function to free the buffer only *after* the callback function has returned. This seeming contradiction can be easily handled by the following code.

```
TOPC_BUF GenerateTaskInput() {
  static void *buf = NULL;
  if ( buf == NULL ) { malloc(buf_size); }
  ... [ Add new message data to buf ] ...
  return TOPC_MSG_PTR(buf, buf_size);
}
```

If buf_size might vary dynamically between calls, the following fragment solves the same problem.

```
TOPC_BUF GenerateTaskInput() {
  static void *buf = NULL;
```

```
    if ( buf != NULL ) { free(buf); }
    ... [ Compute buf_size for new message ] ...
    buf = malloc( buf_size );
    ... [ Add new message data to buf ] ...
    return TOPC_MSG_PTR(buf, buf_size);
  }
```

Note that `buf` is allocated as a *static* local variable. 'TOP-C' restricts the *buf* of `TOPC_MSG_PTR(`*buf*`,` *buf_size*`)` to point to a buffer that is in the heap (not on the stack). Hence, *buf* must *not* point to non-static local data.

8.3.4 Marshaling Complex Data Structures into 'TOP-C' Task Buffers

If you use a distributed memory model and the buffer pointed to by `input` includes fields with their own pointers, the application must first follow all pointers and copy into a new buffer all data referenced directly or indirectly by `input`. The new buffer can then be passed to `TOPC_MSG()`. This copying process is called *marshaling*. See Section 3.1.3 [Marshaling and Heterogeneous Architectures], page 5.

If following all pointers is a burden, then one can load the application on the master and slaves at a common absolute address, and insure that all pointer references have been initialized before the first call to `TOPC_master_slave()`. In 'gcc', one specifies an absolute load address with code such as:

```
gcc -Wl,-Tdata -Wl,-Thex_addr ...
```

These flags are for the data segment. If the pointers indirectly reference data on the stack, you may have to similarly specify stack absolute addresses. Choosing a good *hex_addr* for all machines may be a matter of trial and error. In a test run, print out the absolute addresses of some pointer variables near the beginning of your data memory.

Specifying an absolute load address has many risks, such as if the master and slaves use different versions of the operating system, the compiler, other software, or different hardware configurations. Hence, this technique is recommended only as a last resort.

8.4 Optimizing TOP-C Code for the Shared Memory Model

The 'TOP-C' programmer's model changes slightly for shared memory. With careful design, one can use the same application source code both for distributed memory and shared memory architectures. Processes are replaced by threads. `UpdateSharedData()` is executed only by the master thread, and not by any slave thread. As with distributed memory, `TOPC_MSG()` buffers are copied to 'TOP-C' space (shallow copy). As usual, the application is responsible for freeing any application buffers outside of 'TOP-C' space. Furthermore, since the master and slaves share memory, 'TOP-C' creates the slaves only during the first call to master_slave. If a slave needs to initialize any private data (see `TOPC_thread_private`, below), then this can be done by the slave the first time that it gains control through `DoTask()`.

Two issues arise in porting a distributed memory 'TOP-C' application to shared memory.

1. reader-write synchronization: `DoTask()` must not read shared data while `UpdateSharedData()` (on the master) simultaneously writes to the shared data.

2. creating thread-private (unshared) global variables:

Most 'TOP-C' applications for the distributed memory model will run unchanged in the shared memory model. In some cases, one must add additional 'TOP-C' code to handle these additional issues. In all cases, one can easily retain compatibility with the distributed memory model.

8.4.1 Reader-Writer Synchronization

In shared memory, 'TOP-C' uses a classical single-writer, multiple-reader strategy with writer-preferred for lock requests. By default, `DoTask()` acts as the critical section of the readers (the slave threads) and `UpdateSharedData()` acts as the critical section of the writer (the master thread). 'TOP-C' sets a read lock around all of DoTask() and a write lock around all of UpdateSharedData().

As always in the 'TOP-C' model, it is an error if an application writes to shared data outside of `UpdateSharedData()`. Note that `GenerateTaskInput()` and `CheckTaskResult()` can safely read the shared data without a lock in this case, since these routines and `UpdateSharedData()` are all invoked only by the master thread.

The default behavior implies that `DoTask()` and `UpdateSharedData()` never run simultaneously. Optionally, one can achieve greater concurrency through a finer level of granularity by declaring to 'TOP-C' which sections of code read or write shared data. If 'TOP-C' detects any call to `TOPC_ATOMIC_READ(0)`, 'TOP-C' will follow the critical sections declared by the application inside of `DoTask()` and `UpdateSharedData()`.

`void` **TOPC_ATOMIC_READ** (`0`) { *... C code ...* } Function
`void` **TOPC_ATOMIC_WRITE** (`0`) { *... C code ...* } Function

This sets a global read or write lock in effect during the time that *C code* is being executed. If a thread holds a write lock, no thread may hold a read lock. If no thread holds a write lock, arbitrarily many threads hold a read lock. If a thread requests a write lock, no additional read locks will be granted until after the write lock has been granted. See '`examples/README`' of the 'TOP-C' distribution for example code.

It is not useful to use `TOPC_ATOMIC_READ()` outside of `DoTask()` not to use `TOPC_ATOMIC_WRITE()` outside of `UpdateSharedData()`.

The number 0 refers to page 0 of shared data. 'TOP-C' currently supports only a single common page of shared data, but future versions will support multiple pages. In the future, two threads will be able to simultaneously hold write locks if they are for different pages.

The following alternatives to `TOPC_ATOMIC_READ()` and `TOPC_ATOMIC_WRITE()` are provided for greater flexibility.

`void` **TOPC_BEGIN_ATOMIC_READ** (`0`) Function
`void` **TOPC_END_ATOMIC_READ** (`0`) Function
`void` **TOPC_BEGIN_ATOMIC_WRITE** (`0`) Function
`void` **TOPC_END_ATOMIC_WRITE** (`0`) Function

The usage is the same as for `TOPC_ATOMIC_READ` and `TOPC_ATOMIC_WRITE`.

In the distributed memory model of 'TOP-C', all of the above invocations for atomic reading and writing are ignored, thus retaining full compatibility between the shared and distributed memory models.

8.4.2 Thread-Private Global Variables

A *thread-private* variable is a variable whose data is not shared among threads: i.e., each thread has a private copy of the variable. The only variables that are thread-private by default in shared memory are those on the stack (non-static, local variables). All other variables exist as a single copy, shared by all threads. This is inherent in the POSIX standard for threads in C/C++. If `DoTask()` accesses any global variables or local static variables, then those variables must be made thread-private.

Ideally, if C allowed it, we would just write something like:

```
THREAD_PRIVATE int myvar = 0;  /* NOT SUPPORTED */
```

Instead, 'TOP-C' achieves the same effect '`as if`' it had declared

```
TOPC_thread_private_t TOPC_thread_private;
```

This allows the application writer to include in his or her code:

```
typedef int TOPC_thread_private_t;
#define myvar TOPC_thread_private;
int myvar_debug() {return myvar;} /* needed to access myvar in gdb */
```

'TOP-C' provides primitives to declare a single thread-private global variable. 'TOP-C' allows the application programmer to declare the type of that variable.

TOPC_thread_private Variable

A pre-defined thread-private variable of type, `TOPC_thread_private_t`. It may be used like any C variable, and each thread has its own private copy that will *not* be shared.

TOPC_thread_private_t Type

Initially, undefined. User must define this type using `typedef` if `TOPC_thread_private` is used.

If more than one thread-private variable is desired, define `TOPC_thread_private_t` as a *struct*, and use each field as a separate thread-private variable.

EXAMPLE:

```
/* Ideally, if C allowed it, we would just write:
 *      THREAD_PRIVATE struct {int my_rank; int rnd;} mystruct;
 * We emulate this using TOP-C's implicitly declared thread-private var:
 *      TOPC_thread_private_t TOPC_thread_private;
 */
typedef struct {int my_rank; int rnd;} TOPC_thread_private_t;
#define mystruct TOPC_thread_private
void set_info() {
  mystruct.my_rank = TOPC_rank();
  mystruct.rnd = rand();
}
void get_info() {
foo();
  if (mystruct.my_rank != TOPC_rank()) printf("ERROR\n");
  printf("Slave %d random num:  %d\n", mystruct.my_rank, mystruct.rnd);
```

```
}
TOPC_BUF do_Task() {
  set_info(); /* info in mystruct is NOT shared among threads */
  get_info();
  ...;
}
```

Additional examples can be found by reading 'examples/README' in the 'TOP-C' distribution.

8.4.3 Sharing Variables between Master and Slave and Volatile Variables

The shared memory model, like any 'SMP' code, allows the master and slaves to communicate through global variables, which are shared by default. It is recommended not to use this feature, and instead to maintain communication through `TOPC_MSG()`, for ease of code maintenance, and to maintain portability with the other 'TOP-C' models (distributed memory and sequential). If you do use your own global shared variables between master and slaves, be sure to declare them `volatile`.

```
volatile int myvar;
```

ANSI C requires this qualifier if a variable may be accessed or modified by more than one thread. Without this qualifier, your program may not run correctly.

To be more precise, if a non-local variable is accessed more than once in a procedure, the compiler is allowed to keep the first access value in a thread register and reuse it at later occurrences, without consulting the shared memory. A volatile declaration tells the compiler to re-read the value from shared memory at each occurrence. Similarly, a write to a volatile variable causes the corresponding transfer of its value from a register to shared memory to occur at a time not much later than the execution of the write instruction.

If you suspect a missing volatile declaration, note that 'gcc' support the following command-line options.

```
gcc -fvolatile -fvolatile-global ...
# If topcc uses gcc:
topcc --pthread -fvolatile -fvolatile-global myfile.c
```

The option `-fvolatile` tells 'gcc' to compile all memory references through pointers as volatile, and the option `-fvolatile-global` tells 'gcc' to compile all memory references to extern and global data as volatile. However, note that this implies a performance penalty since the compiler will issue a load/store instruction for each volatile access, and will *not* keep volatile values in registers.

8.4.4 SMP Performance

Note that 'SMP' involves certain performance issues that do not arise in other modes. If you find a lack of performance, please read Section 7.3 [Improving Performance], page 21. Also, note that the vendor-supplied compiler, `cc`, is often recommended over `gcc` for 'SMP', due to specialized vendor-specific architectural issues.

8.5 Modifying TOP-C Code for the Sequential Memory Model

'TOP-C' also provides a sequential memory model. That model is useful for first debugging an application in a sequential context, and then re-compiling it with one of the parallel 'TOP-C' libraries for production use. The application code for the sequential library is usually both source and object compatible with the application code for a parallel library. The sequential library emulates an application with a single 'TOP-C' library.

The sequential memory model emulates an application in which 'DoTask()' is executed in the context of the single slave process/thread, and all other code is executed in the context of the master process/thread. This affects the values returned by `TOPC_is_master()` and `TOPC_rank()`. In particular, conditional code for execution on the master will work correctly in the sequential memory model, but the following conditional code for execution on the slave will probably *not* work correctly.

```
int main( int argc, char *argv[] ) {
  TOPC_init( &argc, &argv );
  if ( TOPC_is_master() )
    ...;  /* is executed in sequential model */
  else
    ...;  /* is never executed in sequential model */
  TOPC_master_slave( ..., ..., ..., ...);
  TOPC_finalize();
}
```

8.6 Caveats

IMPORTANT: 'TOP-C' sets `alarm()` before waiting to receive message from master. By default, if the master does not reply in a half hour (1800 seconds), then the slave receives `SIGALRM` and dies. This is to prevent runaway processes in dist. memory version when master dies without killing all slaves. Section 7.4 [Long Jobs], page 22, in order to change this default. If your applications also uses `SIGALRM`, then run your application with `--TOPC-slave-timeout=0` and 'TOP-C' will not use `SIGALRM`.

`GenerateTaskInput()` and `DoTask()` This memory is managed by 'TOP-C'.

The slave process attempts to set current directory to the same as the master inside `TOPC_init()` and produces a warning if unsuccessful.

When a task buffer is copied into 'TOP-C' space, it becomes word-aligned. If the buffer was originally not word-aligned, but some field in the buffer was word-aligned, the internal field will no longer be word-aligned. On some architectures, casting a non-word-aligned field to 'int' or certain other types will cause a bus error.

9 'TOP-C' Raw Interface for Parallelizing Sequential Code

There are instances when tasks are most naturally generated deep inside nested loops. Often, this occurs in parallelizing existing sequential applications. In such circumstances, it may be difficult to re-write the code to create a function `GenerateTaskInput()`, since that would require turning the loops inside out. (If you don't know what this refers to, then you probably don't need the raw interface.)

On a first reading, you may wish to first look at the example for either a 'for' loop or 'while' loop, depending on the type of loop that you are parallelizing. Then return to the formal descriptions of the 'TOP-C' raw functions. This chapter assumes familiarity with the basic concepts of Chapter 3 [Overview], page 3 and Chapter 4 [Writing TOP-C Applications], page 8.

9.1 'TOP-C' raw functions

void **TOPC_raw_begin_master_slave** (*do_task*, *check_task_result*, *update_shared_data*) void **TOPC_raw_end_master_slave** () — Function

This behaves like master_slave, with `TOPC_raw_submit_task_input(input)` serving the role of `GenerateTaskInput()`. The slave blocks inside `TOPC_raw_begin_master_slave()` and executes '`do_task()`' and '`update_shared_data()`' until the master executes `TOPC_raw_end_master_slave()`. At that time, the slave unblocks. The slave does nothing inside `TOPC_raw_end_master_slave()`.

void **TOPC_raw_submit_task_input** (TOPC_BUF *input*) — Function

Invoked by master between `TOPC_raw_begin_master_slave()` and `TOPC_raw_end_master_slave()`; Typical usage is:

```
TOPC_raw_submit_task_input(TOPC_MSG(&input_data,
                                    sizeof(input_data)) );
```

The argument, *input*, corresponds to what would be returned by `GenerateTaskInput()` in the routine `TOPC_master_slave()`. *input* will be processed by `DoTask()` and its siblings, just as in `TOPC_master_slave()`). There can be multiple occurrences of `TOPC_raw_submit_task_input()`.

TOPC_BOOL **TOPC_raw_wait_for_task_result** () — Function

Invoked by master between `TOPC_raw_begin_master_slave()` and `TOPC_raw_end_master_slave()`; If no tasks are outstanding, returns false immediately. Otherwise, it blocks until a task does return. It calls application callback, `CheckTaskResult()`, and then returns true.

9.2 Parallelizing 'for' Loops

Assume that we are parallelizing a code fragment of the following form. The variables `i` and `j` will be the input to `DoTask()`, and any data structures indexed by `i` and `j` (for example `array` in `array[i][j]`) will be part of the shared data.

```
float array[ROWS][COLS];
...
  for ( i = 0; i < 10; i++ ) {
    for ( j = 0; j < 10; j++ ) {
      /* do_task: */ ...
      /* update:  */ array[i][j] = ...;
```

```
    }
  }
```

Assume that the labels do_task and update above correspond to the callback functions DoTask() and UpdateSharedData(). Then the code is parallelized below.

```
float array[ROWS][COLS];
typedef struct {int i_val; int j_val;} input_t;
void *DoTask(input_t *buf) {
  int i = (*buf).i_val, j = (*buf).j_val;
  /* do_task: */ ...
}
void *CheckTaskResult(input_t *buf, output_t *buf2) {
  /* update:  */ array[i][j] = ...;
  return NO_ACTION;
}
main(int argc, char **argv) {
  TOPC_init( &argc, &argv );
  TOPC_raw_begin_master_slave(DoTask, CheckTaskResult,
                              UpdateSharedData);
  if (TOPC_is_master()) {
    for ( i = 0; i < 10; i++ ) {
      for ( j = 0; j < 10; j++ ) {
        input_t input;
        input.i_val = i; input.j_val = j;
        TOPC_raw_submit_task_input( TOPC_MSG(&input, sizeof(input)) );
      }
    }
  } TOPC_raw_end_master_slave();
  TOPC_finalize();
}
```

9.3 Parallelizing 'while' Loops

Assume that we are parallelizing a code fragment of the following form and input is a pointer.

```
while ( (input = next_input()) != NULL ) {
  /* do_task: */ ...
  /* update:  */ ...
}
```

Assume that the labels do_task and update above correspond to the callback functions DoTask() and UpdateSharedData(). Then the code is parallelized below, where input_size must be specified by the application before it is used.

```
TOPC_init( &argc, &argv );
TOPC_raw_begin_master_slave(DoTask, CheckTaskResult,
                            UpdateSharedData);
if (TOPC_is_master()) {
  while ( (input = next_input()) != NULL
          || TOPC_raw_wait_for_task_result() ) {
    TOPC_raw_submit_task_input( TOPC_MSG(input, input_size) );
```

```
    }
  } TOPC_raw_end_master_slave();
  TOPC_finalize();
```

Note that the code inside the *raw begin/end block* is executed only by the master in the code above.

If the buffer, `input`, contains pointers to other data, then you will need to *marshal* the data before calling `TOPC_MSG()`. See Section 3.1.3 [Marshaling and Heterogeneous Architectures], page 5.

10 Acknowledgements

A project of this scope cannot be achieved alone. While Gene Cooperman was the primary author, the project has benefited from contributions by several various individuals and institutions at different times.

The author wishes to thank the National Science Foundation for support under which much of this work was developed. The author wishes to thank the Mariner Project at Boston University for the use of the Origin 2000 and other facilities which helped in the development of this software. An earlier, experimental version of mpinu (MPI subset) was written by Markos Kyzas and partially revised by Gene Cooperman. Michael Weller provided ideas for improving some of the C code, and provided valuable feedback when he adapted the '`TOP-C`' ideas to a large application on an IBM SP-2. The loader module is joint with Victor Grinberg. Further experience and feedback was gained from the GAP community when the '`TOP-C`' model was ported to ParGAP, a refereed share package. (GAP – Groups, Algorithms and Programming) is a language similar to Maple, specialized for symbolic computations in computational algebra and especially computational group theory.) '`TOP-C`', version 2, was exported by Victor Grinberg from ParGAP, with enhancements by Gene Cooperman. Some important feedback was gained in the TOP-C parallelization of Geant4. (Geant4 is a toolkit for the Monte Carlo simulation of particle-matter interaction. The package has close to a million lines of C++ code. The TOP-C parallelization is included with the Geant4 distribution.) Xiaoqin Ma analyzed mechanisms for detecting and recovering from dead slaves, broken sockets, etc., and wrote the first version of code to handle that.

Appendix A Summary of 'TOP-C' Commands

From Section 4.1 [The Main TOP-C Library Calls], page 8.

`void TOPC_init ( int *argc, char ***argv )`
`void TOPC_finalize ( void )` Function

```
void TOPC_master_slave
        ( TOPC_BUF (*generate_task_input)(),
          TOPC_BUF (*do_task)(void *input),
          TOPC_ACTION (*check_task_result)(void *input, void *output),
          void (*update_shared_data)(void *input, void *output)
        )
```
Function

`TOPC_BUF TOPC_MSG ( void *buf, int buf_size )` Function

From Section 4.2 [Callback Functions for `TOPC_master_slave()` *], page 9.*

`TOPC_BUF GenerateTaskInput ( void )` Function

`TOPC_BUF DoTask ( void *input )` Function

`TOPC_ACTION CheckTaskResult ( void *input, void *output)` Function

`void UpdateSharedData ( void *input, void *output )` Function

From Section 4.3 [Actions Returned by `CheckTaskResult()` *], page 10.*

`Action TOPC_ACTION NO_ACTION` Action

`Action TOPC_ACTION UPDATE` Action

`Action TOPC_ACTION REDO` Action

`Action TOPC_ACTION CONTINUATION ( void *next_input )` Action

From Section 4.4 [TOP-C Utilities], page 10.

`TOPC_BOOL TOPC_is_up_to_date ( void )` Function

`int TOPC_rank ( void )` Function

`TOPC_BOOL TOPC_is_master ( void )` Function

`int TOPC_num_slaves ( void )` Function

`int TOPC_num_idle_slaves ( void )` Function

`int TOPC_node_count ( void )` Function

From Section 8.4 [Optimizing TOP-C Code for the Shared Memory Model], page 26.

`TOPC_thread_private` Variable

`TOPC_thread_private_t` Type

`void TOPC_ATOMIC_READ ( 0 ) {` *... C code ...* `}`
`void TOPC_ATOMIC_WRITE ( 0 ) {` *... C code ...* `}` Function

`void TOPC_BEGIN_ATOMIC_READ ( 0 )`
`void TOPC_END_ATOMIC_READ ( 0 )`
`void TOPC_BEGIN_ATOMIC_WRITE ( 0 )`
`void TOPC_END_ATOMIC_WRITE ( 0 )` Function

From Chapter 9 [Raw 'TOP-C' interface: `raw_master_slave` *], page 30.*

`void TOPC_raw_begin_master_slave`
(*do_task*, *check_task_result*, *update_shared_data*)

`void TOPC_raw_end_master_slave ()` Function

`void TOPC_raw_submit_task_input ( TOPC_BUF` *input* `)` Function

`TOPC_BOOL TOPC_raw_wait_for_task_result ()` Function

From Section 8.2 [Aborting Tasks], page 23.

`void TOPC_abort_tasks ( void )`
`TOPC_BOOL TOPC_is_abort_pending ( void )` Function

From Section 8.1 [Testing for Task Continuations], page 23.

`TOPC_BOOL TOPC_is_REDO ( void )`
`TOPC_BOOL TOPC_is_CONTINUATION ( void )` Function

From Section 8.3.2 [Large Buffers and TOPC_MSG_PTR], page 24.

`TOPC_BUF TOPC_MSG_PTR ( void *`*buf*`, int` *buf_size* `)` Function

From Section 5.2 [Command Line Options in TOP-C Applications], page 12.

`--TOPC-help‘[=<0/1>]’`
[boolean, default: false]

`--TOPC-verbose‘[=<0/1>]’`
[boolean, default: false]

`--TOPC-stats‘[=<0/1>]’`
[boolean, default: false]

`--TOPC-num-slaves‘=<int>’`
[default: -1 (system-defined)]

`--TOPC-slave-wait‘=<int>’`
[default: 0]

`--TOPC-slave-timeout‘=<int>’`
[default: 1800 s]

`--TOPC-trace‘=<int: 0/1/2>’`
[trace (0: notrace, 1: trace, 2: user trace fncs, default: 2)]

`--TOPC-procgroup‘=<string>’`
[default: "./procgroup"]

`--TOPC-safety‘=<int: 0..20>’`
[default: 0]

From Section 6.4 [Tracing Messages], page 19.

```
int TOPC_OPT_trace
void (*)(void *input) TOPC_OPT_trace_input
void (*)(void *input, void *output) TOPC_OPT_trace_result
```

Appendix B Example 'TOP-C' Application

There are several example 'TOP-C' programs in the 'topc/examples' subdirectory. We include one example in this manual. It does not contain any UPDATE actions, and therefore illustrates only a trivial form of parallelism (with no interaction among the slaves). The 'topc/examples' subdirectory should be inspected for more sophisticated examples. After understanding this example, you may also want to look at Chapter 8 [Advanced Features], page 23, or if you are parallelizing a sequential program, then you may want to look at Chapter 9 [TOP-C Raw Interface], page 30.

This program produces an array of 10,000,000 random integers in one pass, and then finds the maximum value in a second pass. It would be compiled by: `topcc` *MODE* `'file.out'`, where *MODE* is one of `--seq`, `--mpi`, or `--pthread`. One can control the number of slaves by executing: `./a.out --TOPC-num-slaves=`*num*.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <topc.h>

#define MAX 2000000
#define INCR MAX/10 /* We assume INCR divides MAX exactly */

int array[MAX];
int idx;
int max_int;

TOPC_BUF GenerateTaskInput() {
  int input_task;
  if (idx >= MAX) return NOTASK;
  input_task = idx;
  idx = idx + INCR;
  return TOPC_MSG( &input_task, sizeof(input_task) );
}
TOPC_BUF DoTaskRandom( int *ignore ) {
  int rand_int[INCR];
  int i;
  for ( i = 0; i < INCR; i++)
    rand_int[i] = rand();
  return TOPC_MSG( rand_int, INCR * sizeof(int) );
}
TOPC_ACTION CheckTaskRandom( int *input, int rand_vals[] ) {
  int curr_idx = *input;
  memcpy( array+curr_idx, rand_vals, INCR * sizeof(int) );
  return NO_ACTION;
}

TOPC_BUF GenerateTaskInMax() {
  int *input_task;
  if (idx >= MAX) return NOTASK;
```

```
  input_task = array + idx;
  idx = idx + INCR;
  return TOPC_MSG( input_task, INCR * sizeof(int) );
}
TOPC_BUF DoTaskMax( int subarray[] ) {
  int i;
  int max=0;
  for ( i = 0; i < INCR; i++)
    if ( subarray[i] > max )
      max = subarray[i];
  return TOPC_MSG( &max, sizeof(max) );
}
TOPC_ACTION CheckTaskMax( int ignore[], int *output ) {
  int curr_max = *output;
  if ( curr_max > max_int )
    max_int = curr_max;
  return NO_ACTION;
}

int main( int argc, char **argv ) {
  /* Set default to no trace; Override with:  ./a.out --TOPC-trace=1 */
  TOPC_OPT_trace = 0;
  TOPC_init( &argc, &argv );
  idx = 0; /* Initialize idx, and randomize values of array[] */
  TOPC_master_slave(GenerateTaskInput, DoTaskRandom, CheckTaskRandom,
                    NULL);
  if (TOPC_is_master())
    printf("Finished randomizing integers.\n");
  idx = 0; /* Re-initialize idx to 0, and find max. value in array[] */
  TOPC_master_slave( GenerateTaskInMax, DoTaskMax, CheckTaskMax, NULL );
  TOPC_finalize();
  printf("The maximum integer is:  %d\n", max_int);
  exit(0);
}
```

Appendix C Using a Different 'MPI' with TOP-C

'TOP-C' provides a built-in subset of MPI sufficient to run distributed applications. If you prefer to use a different implementation of 'MPI' with 'TOP-C', this note describes how to do so. Examples for 'MPICH', 'LAM' and 'IBM POE' are at the end of this section. You should re-configure and re-build 'TOP-C' using the same C/C++ compiler as that of your chosen MPI.

First test a 'hello_world' MPI program for your chosen MPI implementation to make sure that the startup mechanism is well understood.

Many dialects of 'MPI' provide their own wrapper around the C and C++ compiler. For example, 'LAM MPI' provides `mpicc`. If your dialect provides such a wrapper `mpicc`, then do `./configure --with-mpi-cc=mpicc` and `make`. There is a similar configure flag, `--with-mpi-cxx`, for 'C++'.

If you do use such a wrapper, you should use the same C compiler for the rest of 'TOP-C'. Hence, if `mpicc` uses `cc`, then configure with `env CC=cc ./configure --with-mpi-cc=mpicc --cache-file=/dev/null`. In general, you can always specify a non-default C and C++ compiler by specifying `CC=...` and `CXX=...`, respectively.

If your dialect does not provide such a wrapper, it is easy to create one by modifying the shell script below for your site.

```
#!/bin/sh
gcc -I/usr/local/include $* -L/usr/local/lib -lmpi
```

These wrappers will enable `topcc` and `topc++` to work, but not `topc-config`. If you also need `topc-config`, then you must modify `topc-config`. Determine the libraries used by your dialect of MPI. Then search for a string with `-ltopc-mpi` in '.../topc/bin/topc-config', and append to it your 'libmpi'. For example, append to `-ltopc-mpi` with `-L/usr/local/lib -lmpi`.

Finally, for many dialects of MPI, after compiling the MPI application, they may also require a special command at runtime to execute, such as `mpirun a.out`. In such cases, your 'TOP-C' application must be treated similarly.

The following examples illustrate the configuration and use of MPI for other MPI implementations using 2 slaves and 1 master.

```
# IBM POE/AIX:
  env CC=xlc CXX=xlC ./configure --with-mpi-cc=mpcc --with-mpi-cxx=mpCC
  make; cd bin; ./topcc --mpi -qcpluscmt ../examples/parfactor.c
  poe ./a.out 1234 --TOPC-stats -procs 3 -pgmmodel spmd

# MPICH/Linux:
  ./configure --with-mpi-cc=mpicc
  make; cd bin; ./topcc --mpi ../examples/parfactor.c
  mpirun -np 3 ./a.out --TOPC-stats 1234

# LAM/Linux:
  ./configure --with-mpi-cc=mpicc
  make; cd bin; ./topcc --mpi ../examples/parfactor.c
  mpirun -c 3 ./a.out -- --TOPC-stats 1234
```

Function Index

Variable Index

Concept Index

Short Contents

Table of Contents