Task Oriented Parallel C/C++: A Tutorial (Version 0.92)

Gene Cooperman
gene@ccs.neu.edu

October 25, 2003

1 Introduction

The goal of Task Oriented Parallel C/C++ (TOP-C) is to provide a utility that makes it easy to take existing sequential code and parallelize it. The TOP-C constructs allow the end user to parallelize a sequential program while modifying relatively few lines of his or her original source code. This is useful for very large programs, and for sequential programs that are likely to pass through frequent version changes.

TOP-C runs under most dialects of UNIX/Linux. It is free and open-source software. Its home page is at `http://www.ccs.neu.edu/home/gene/topc.html`. It includes a 40-page manual. This tutorial is intended as a gentler introduction to TOP-C.

2 A Quick Introduction to TOP-C

TOP-C uses a master slave architecture. This is a virtual architecture that may map onto a distributed, shared memory, or other architecture. A single TOP-C application can be linked to the TOP-C libraries using either a distributed memory model (message-based, for clusters) or a shared memory model (thread-based, for a multiprocessor node), simply by linking with the appropriate TOP-C library. TOP-C also supports a sequential memory model (emulating TOP-C in a single process for ease of debugging), and a Grid memory model (based on the Globus protocols). For purposes of definiteness, we use the language of distributed memory.

2.1 A First Look at the Software Model

TOP-C uses a master slave architecture (see Figure 1). When parallelizing sequential code using TOP-C, one should first identify the inner procedure where most of the CPU time of the sequential application is spent. (Sometimes this inner procedure may instead be a certain loop construct, or it may consist of multiple inner procedures. There are easy ways to capture these variations within TOP-C.)

This CPU-intensive inner procedure is modelled as a task (see Figure 2). The task input represented as the procedure arguments, and the task output represented as the return value. Accesses to global variables are modelled as access to shared data (since multiple slave processors executing the same procedure may access the same global variable). Updates to the global variables are postponed in TOP-C, and must be processed via the master process (as will be explained later in Sections 2.3.2 and 3).

Intuitively, TOP-C will execute each task (each call to the inner procedure) on a distinct slave process. As slave process become available, TOP-C computes a new task input on the master, and passes it to the slave process. The task output is computed on the slave process, and returned to the master. The precise handling of the shared data (global variables) is explained in 2.3.3.

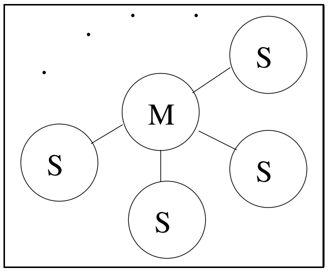

Figure 1: TOP-C master-slave star topology

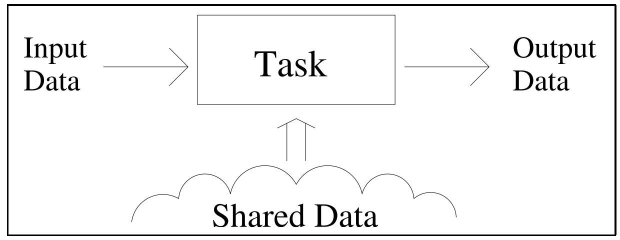

Figure 2: TOP-C Concept 1: The Task

In order to model this CPU-intensive inner procedure as a task, the application writer must write four callback functions that describe the task. These callback functions are short, and act as glue between the original sequential application and the TOP-C parallel library.

The manner by which a TOP-C application models its inner procedure by a task is summarized in Figure 3. Specifically, when TOP-C discovers an available slave processor, it calls the application routine [4] `GenerateTaskInput()` on the master, to derive the next task input. This task input is passed across the network to the slave process. On the slave, TOP-C calls the application routing `DoTask()`, which executes the CPU-intensive inner procedure and produces the task output. TOP-C passes the task output back to the master process. Within the master process, TOP-C presents the input-output pair computed on the slave, and calls `CheckTaskResult(input, output)` to determine what action to take. If the application involves trivial parallelism, the routine `CheckTaskResult()` will typically record the result and return "no further action". On the other hand, if there is a need to modify a global variable, `CheckTaskResult()` will return `UPDATE`, and TOP-C will call `UpdateSharedData(input, output)` on each process.

```
1. Master:  GenerateTaskInput() ⇒ task_input
2. Slave:  DoTask(task_input) ⇒ task_output
3. Master:  CheckTaskResult(task_input, task_output) ⇒ TOPC_action
4. Everywhere:  UpdateSharedData(task_input, task_output)
```

Figure 3: Four User-Defined Callback Functions

Note that the names of the four callback functions are merely a convention used in this exposition. As we will see in Figure 4, the callback functions are declared to TOP-C in Figure 4, and can be called by any names convenient to the application writer.

2.2 A First Look at Writing a TOP-C Application

A TOP-C application consists of application code that is linked to the TOP-C library. A typical application has the structure of Figure 4. The job of `TOPC_init()` is to create child processes (or additional threads, if we are operating in a shared memory environment). The child processes execute the same code as the master, in the typical style of Single Program Multiple Data (SPMD). Eventually, all processes reach the portion of the code containing the primary TOP-C system call, `TOPC_master_slave()`. It is here that the four application-defined callback functions of the previous section are declared to TOP-C.

Once `TOPC_master_slave()` is called, control is passed to the TOP-C library. The TOP-C library

```
#include "topc.h"

main(int argc, char **argv) {
  TOPC_init(&argc, &argv);
  ...
  TOPC_master_slave(GenerateTaskInput,
                    DoTask,
                    CheckTaskResult,
                    UpdateSharedData);
  ...
  TOPC_finalize();
}
```

Figure 4: Invoking TOP-C

then invokes the four user-defined TOP-C callback functions as described earlier. Upon completion, control returns to the code after `TOPC_master_slave()`. The end user may optionally add further calls to `TOPC_master_slave()`. After `TOPC_finalize()`, the slave processes then exit.

2.3 Three Concepts for TOP-C

The callback functions can best be understood by considering three TOP-C concepts:

1. The Task (tasks executed by slave in parallel)
2. The Action (directing the parallel strategy for TOP-C)
3. The Shared Data (shared and readable by all processes)

2.3.1 The Task

The task is a function that takes a single input parameter, and returns a single output parameter. The task may also read the shared data. The task may *not* write to the shared data. This was illustrated in Figure 2. The task input and output are considered by TOP-C to simply be buffers of bits of type `(void *)`.

Note that TOP-C has no conception of the possibly complex user data types being passed as the task input and task output. Both the task input and the task output are declared to TOP-C as type `TOPC_BUF`, which is a synonym for `void *`. This is intentional, as it helps to maintain the simplicity of TOP-C. The application writer never needs to declare data types to TOP-C. Declaration of complex data types is an inherent part of such systems as MPI, PVM, and Corba. Complex declarations of in major applications are often responsible for a steep learning curve for the new user.

User data must be *serialized* (or *marshaled*) into such a buffer in order to pass data across the network to a different CPU. This responsibility for serialization is entirely a user responsibility. In spirit, this division of labor is similar to the C and C++ choices to make I/O routines a library independent of the core language. Many other facilities exist to aid the user in serialization. The system calls `ntohl` and `htonl` ensure portability of data of type `int`. The IEEE floating point standard ensures compatibility in serializing floating point. The package `XDR` is available for portability of the more complex data structures between heterogeneous systems. The author considers the issue of easy serialization of complex data structures to be a continuing area of research, which should not be directly tied to TOP-C.

2.3.2 The TOP-C Action

The TOP-C action is the place where the application writer implements his or her parallel strategy. The user callback function, `CheckTaskResult(task_input, task_output)`, examines its arguments and then returns one of the following three actions.

1. `NO_ACTION` (for trivial parallelism)
2. `UPDATE` (one slave discovers information useful for many slaves)
3. `REDO` (arbitrating among multiple slave updates)

The action `NO_ACTION` is typically returned in the case of trivial parallelism. Since `CheckTaskResult()` is executed on the master, the master process can save a copy of its argument pair, (`task_input`, `task_output`), in a table on the master process pointed to by a global variable.

The action `UPDATE` is used to update the shared data. Recall from Section 2.3.1 that the shared data is treated as read-only data by the task. Hence, the `UPDATE` action is the only mechanism to update the shared data. Any non-trivial parallel strategy by which output data from one slave is passed to other slaves must proceed by means of an `UPDATE` action. More about this is contained later in Section 2.3.3.

It may happen that a slave is already executing a task (executing `DoTask()` when an `UPDATE` request arrives from the master. In such cases, the slave does not abort the current task or preempt it. The slave completes the current task, returns a corresponding `task_output` to the master, and only then recognizes the `UPDATE` request from the master.

In the previous situation of an `UPDATE`, a common TOP-C strategy is to ask the same slave executing the original task to re-execute that task after the shared data has been updated. This is only one of several possible parallel strategies in TOP-C. More on the implementation of non-trivial parallel strategies for TOP-C and the use of the `REDO` action are contained in Section 3.

The action `REDO` tells TOP-C that the original task should be re-executed with the original input and on the *same* slave node on which it was originally executed. This make more advanced parallel strategies such as described in Sections 3.2 and following.

The interaction of the TOP-C actions in TOP-C is summarized in Figure 5 below.

2.3.3 The Shared Data

The shared data is the primary mechanism by which slaves can communicate information to other slaves. In the case of TOP-C running on top of a distributed memory architecture, the shared data is replicated on each node (on the master and on each slaves). The shared data can be read by any routine on any process. But the shared data must be written (modified) only through the routine `CheckTaskResult()`. This is summarized in Figure 6.

It is the responsibility of the application writer to write only through the routine `CheckTaskResult()`. This is an implicit contract with TOP-C. TOP-C has no way of verifying this contract. The results of modifying the shared data outside of a call through `CheckTaskResult()` are unpredictable.

Note that the end user never needs to declare the shared data to TOP-C. It is enough that the end user registers the callback function `CheckTaskResult()` with TOP-C. Since all nodes (master and slaves) execute the same program with the same initial data, they also begin with the same shared data. Whenever the shared data must be updated, TOP-C updates it by calling `CheckTaskResult(task_input, task_output)`, which implements an incremental update to the shared data.

Note that at all times, the master has the most recent copy of the shared data. Furthermore, as noted in Section 2.3.2, `UPDATE` requests from the master are honored by the slave only after the slave completes the

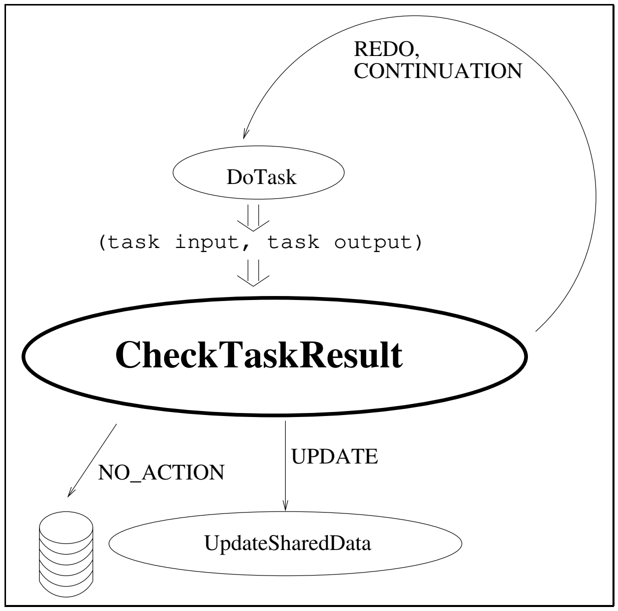

Figure 5: TOP-C Concept 2: Action

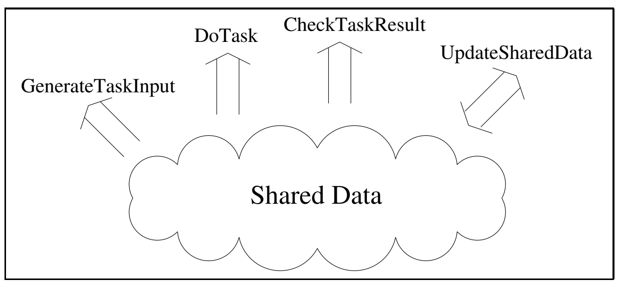

Figure 6: TOP-C Concept 3: Shared Data

current task. Hence, eventually, the shared data on a slave will be in sync with the master, but at any given moment, the slave copy may be waiting on pending updates. Hence, the slave has a policy of *lazy updates*, rather than eager updates.

Note also that in the TOP-C design, the application writer is encouraged to send over the network only the incremental information encapsulated in the result, (`task_input`, `task_output`). This is usually cheaper than copying the entire shared data from the master to each slave process. The user-defined routine `UpdateSharedData()` then uses this incremental information plus the old shared data to construct a newer version of the shared data.

2.4 Other TOP-C Utilities

`TOPC_MSG(void *buf,size_t size);` `TOPC_is_up_to_date()` (see Figure 7 and Figure 8 for examples of usage.)

3 Implementing Parallel Strategies in TOP-C

Given a working sequential application, and a parallel strategy, a working TOP-C application can usually be quickly implemented. For a first look at this statement, consider the issue of writing the four user-defined

```
#include "topc.h"
typedef int OUTPUT_t;
OUTPUT_t foo (int x) { return x+1; }
OUTPUT_t answer_array[10];

TOPC_BUF GenerateTaskInput() {
  static int i = 0;
  i++;
  if (i >= 10) return NOTASK;
  else return TOPC_MSG(&i, sizeof(i));
}
TOPC_BUF DoTask(int *i) {
  OUTPUT_t output = foo(*i);
  return TOPC_MSG(&output, sizeof(output));
}
TOPC_ACTION CheckTaskResult( int *input, OUTPUT_t *output ) {
  answer_array[*input] = *output;
  return NO_ACTION;
}
void UpdateSharedData ( int *input, OUTPUT_t *output ) { }

int main( int argc, char **argv ) {
  TOPC_init( &argc, &argv );
  TOPC_master_slave(GenerateTaskInput,
                    DoTask,
                    CheckTaskResult,
                    UpdateSharedData);
  TOPC_finalize();
}
```

Figure 7: Implementing Trivial Parallelism: Executing `foo()` on multiple slaves;
In this examp[le, foo computes squares of integers.

TOP-C callback functions for a trivial parallel application.

3.1 Implementation of Trivial Parallelism

The first parallelization task is to choose the tasks to be executed in parallel on different CPUs. This task generally corresponds to the body of some loop (possibly an inner loop) of the sequential code, where most of the work is done. The code corresponding to the task will become the body of the function `DoTask()` in the parallelized code.

As an example, we consider the trivial parallel program, as given in Figure 7.

3.2 Implementation of Optimistic Concurrency for Parallelism

A second strategy easily implemented in TOP-C is one of *optimistic concurrency*. In databases, a policy of optimistic concurrency means that a node executes a transaction without committing the result. A check is then made whether the two concurrent transactions were legitimately executed in parallel. If the two transactions could not be legitimately executed in parallel, then one transaction is committed, while the second transaction is rolled back, and then re-done. Figure 8, below, demonstrates the code to implement optimistic concurrency.

The TOP-C action `REDO`, and the TOP-C utility `TOPC_is_up_to_date()` are all that are needed to implement optimistic concurrency. The utility `TOPC_is_up_to_date()` returns 1 or 0 (true or false).

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

Figure 8: Non-Trivial Parallelism: A Simple Idiom for TOP-C

It checks whether the last task returned to the master was done on a slave whose shared data is in sync with the master. If it was so, then the shared data of that slave is *up to date*, and the utility returns 1. If the master has sent an `UPDATE` request to the slave after the master sent the task input for the last task, then `TOPC_is_up_to_date()` returns 0 (false). It is easy for the master to decide whether a recent task is up to date by keeping timestamps for all task inputs sent out, and for all `UPDATE` requests sent out.

In Figure 8, if we're lucky, then `REDO`s are rare, and the program is efficient. This is the essence of optimistic concurrency. Even if `REDO`s are common, the implementation in Figure 8 remains correct. Making such code efficient is the subject of the next section.

3.3 Implementation of General Parallel Strategies

Even though `REDO`s may be common, the callback functions `DoTask()` and `UpdateSharedData()` can be modified to yield an efficient implementation. To understand how, we examine in Table 1 the sequence of events as seen by a slave process.

Slave1	Slave2	...
input1	input2	
`DoTask()`		
output1	`DoTask()`	
`UPDATE1`	output2	
`UpdateSharedData()`	`UPDATE1`	
	`UpdateSharedData()`	
	`REDO (input2)`	
	`DoTask()`	

Table 1: `REDO` action from viewpoint of Slave2

In this scenario, Slave2 has received a `REDO` request from the master. Table 1 labels this action as `REDO` (input2), to remind us that the `REDO` action derives from examining the result (input2, output2) of the task given to Slave2.

The master had previously sent task inputs, input1 and input2, to Slave1 and Slave2, respectively. Each slave sent the result back to the master. The master saw first the result of Slave1. As a result, the master sent an `UPDATE` request (`UPDATE1`) to all processes. The master then saw the result of slave2, (input2, output2). In keeping with the logic of figure 8, the master then sent a `REDO` request (`REDO2`) to Slave2.

This scenario illustrates precisely the inefficiency issue about which we are worried. Slave2 is executing the task input2 twice. However, if the implementation of `DoTask()` and `UpdateSharedData()` were just a little more clever, the second execution of the task for input2 would be much faster.

The trick is that when Slave2 is executing `DoTask()` and `UpdateSharedData()`, Slave2 should be "taking notes" on the intermediate data generated in those computations. An easy way to take notes is to copy intermediate data to global variables. Later, the `REDO` request will cause Slave2 to re-execute

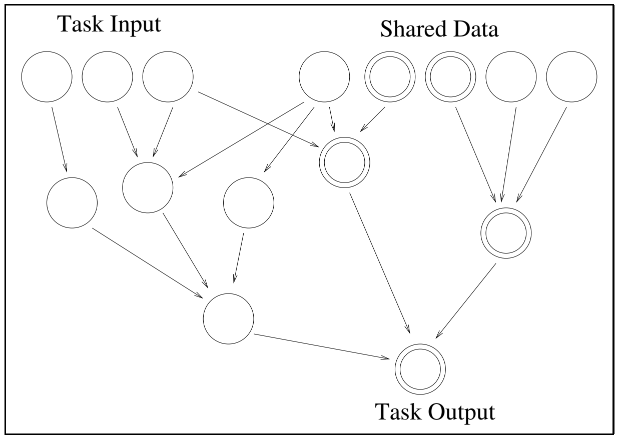

Figure 9: `DoTask()` viewed as a Data Dependency computation(double circles indicate modified data)

`DoTask()` with the original input data, input2, but with modified shared data. By using the "notes" from the original execution, Slave2 can then avoid re-computing some of the intermediate quantities.

In a sense, the task computation (see Figure 2) can be viewed as a data-flow computation proceeding from the task input and the shared data. When the shared data data is altered, one traces the data flow to determine which intermediate computations must be re-computed.

Of course, there is no application-independent way to describe which portions of the original task execution need to be re-done, and which portions can be copied from the "notes". But this is the way it should be. In the early days of optimizing compilers, a favorite joke was that it is not the job of an optimizing compiler to compile a bubble sort algorithm into quicksort.

In the same vein, it is not the job of TOP-C to decide which intermediate data should be copied over into global variables, in case of re-execution due to a `REDO`. This is the heart of the parallel strategy. Nevertheless, the methodology of this section makes it obvious to the application writer how to develop a reasonable parallel strategy for each application.

Visually, one can imagine this strategy by the diagram in Figure 9. Imagine the computations of `DoTask()` as data dependency computations. The data at each node is computed from previous nodes. After a `UPDATE`, some of the shared data is modified (indicated by double circles). The intermediate and final data depending on the modified data is also indicated by double circles. During a `REDO`, `DoTask()` need only re-compute the data of the double circles, if the application writer had previously saved the intermediate node data computed during the original `DoTask()`, and if the application writer had recorded which nodes of the shared data were modified during `UpdateSharedData()`.

4 Usage of TOP-C

4.1 Installation

TOP-C can be downloaded from the TOP-C home page, `http://www.ccs.neu.edu/home/gene/topc.html`. Once you have downloaded it, say as topc.tar.gz, it remains to do the following.

```
cp topc.tar.gz INSTALL_DIRECTORY/
cd INSTALL_DIRECTORY
gzip -dc topc.tar.gz | tar xvf -
cd topc
./configure
```

```
make
```

That's it. If you like, you can test TOP-C "out of the box", by typing `make test`.

In order to use TOP-C in distributed memory mode, it must be possible for the master to create remote slave processes on the slave hosts without being asked for a password. To verify this, try
`rsh` *slave_host*

If your operating system does not allow `rsh` or if it asks you for a password, then try the newer package, `ssh`. If `ssh` *slave_host* causes your operating system to ask you for a password, and if you are using a shared file system among all your hosts, then you can fix it as follows:

```
ssh-keygen -t dsa
  [ accept all default values; Press <RETURN> in answer to all questions. ]
ssh-keygen -t rsa
  [ accept all default values; Press <RETURN> in answer to all questions. ]
cd ~/.ssh/
cat id*.pub >> authorized_keys
```

Now, `ssh` should allow you to login between hosts on the shared file system without a password.

4.2 Compiling a TOP-C Application

TOP-C provides a wrapper, `topcc`, for the default C compiler (and `topc++` for the default C++ compiler). These scripts are located in `INSTALL_DIRECTORY/topc/bin/`. The wrapper `topcc` takes the same arguments as the default C compiler, and it additionally recognizes command line arguments specifying for what communication model (memory model) it should be compiled. TOP-C is implemented in a layered style and can be based on any of several communication models: MPI, POSIX threads, the Globus Grid protocols, etc. Writing a new communication layer for TOP-C is the job of a single day.

In the example below, note that `topcc` passes on to the default C compiler any options not intended for TOP-C. In this example, `topcc` is compiling for the MPI communication layer (distributed memory model) with C options `-g -O0 -o myapp`. TOP-C uses its own built-in MPI package by default, although TOP-C also provides a configure option to substitute an MPI of the user's choice.

```
topcc --help
topcc --mpi -g -O0 -o myapp myapp.c
./myapp
```

Other communication layers include `--seq` for sequential memory (single process), and `--pthread` for POSIX threads (shared memory model). Debugging a TOP-C application first under sequential memory makes it much easier to later catch any remaining distributed memory bugs.

In the case of a distributed memory model such as MPI, the user must specify the host for each slave process, along with the directory and name of the application binary on the remote host. This is done by specifying a `procgroup` file (by default, in the current directory). Where possible, TOP-C runs the slave process in a directory with the same name as the current directory on the master.

In the example procgroup file of Figure 10, the argument "-" for hostA means that the original command line is repeated in executing the slave process. The "-" on the command lines for hostB means that the command line arguments of the master process is repeated for the slave process. The first three slave processes have their standard output redirected to files, while the fourth and fifth slave processes send their standard output to the user's terminal. The writer of this procgroup file is intentionally running tow slave

```
local 0
hostA 1 - > slave1.out
hostB 1 myapp - > slave2.out
hostB 1 myapp - > slave3.out
hostC 1 /net/hostA/home/joe/myapp -
hostC 1 /net/hostA/home/joe/myapp -
```

Figure 10: procgroup: specifying remote hosts for the slave processes

processes on the same host in the cases of hostB and hostC. This has the benefit of allowing one slave process to computer while the other waits for communication. Hence, this wastes fewer slave idle cycles, when the number of slave processors is limited.

4.3 Execution

```
./myapp --TOPC-help
  INCLUDE:  --TOPC-num-slaves=...
            --TOPC-procgroup=...
             etc.
./myapp 123
./myapp --TOPC-trace=0  123
```

FILL IN