TOP-C: Task-Oriented Parallel C for Distributed and Shared Memory

Gene Cooperman*

College of Computer Science
Northeastern University
Boston, MA 02115
gene@ccs.neu.edu

Summary. The "holy grail" of parallel software systems is a parallel programming language that will be as easy to use as a sequential one, while maintaining most of the potential efficiency of the underlying parallel hardware. TOP-C (Task-Oriented Parallel C) attempts such a model by presenting a task abstraction that hides much of the details of the underlying hardware. DSM (Distributed Shared Memory) also attemps such a model, but along an orthogonal direction. By presenting a shared memory model of memory, it hides much of the details of message-passing required by the underlying hardware. This article reviews the TOP-C model and then presents ongoing research on combining the advantages of both models in a single system.

1. Introduction

This paper proposes the TOP-C model as a way to easily organize computations on DSM systems with many processors, while maintaining high concurrency. The proposed model allows the application writer to implicitly declare segments of his environment that correspond to the program objects that he is using. The segments are implicit in that the application writer need only declare to TOP-C which segments are modified by a given routine.

TOP-C has been successful in executing many large, parallel applications [4, 8, 10, 11, 12, 17]. TOP-C is implemented as a C library, and does not require a modification of the programming language of the application. As with any C library, the TOP-C library can also be used by a C++ program. One can choose any of three TOP-C libraries to choose between: SMP (Symmetric MultiProcessing, or shared memory) architectures, distributed memory architectures, and a sequential architecture. The application writer may continue to use his or her favorite programming language as long as that language has an interface to C libraries.

It should be noted that current high-end SMP architectures (many processors) are quite similar to DSM systems with hardware support. Hence, there appears to be a gradual progression from low-latency SMP through medium-latency DSM systems, with no sharp dividing line. Accordingly, we talk about the SMP version of the TOP-C model with the intention that this also applies to DSM.

* Supported in part by NSF Grant CCR-9732330.

Section 2. describes the TOP-C model. Section 3. then motivates why the model needs to be extended when the environment uses a lot of memory. Section 4. then describes a natural way to enhance the TOP-C model by providing an application abstraction of *segments*. If the application program is an object-oriented C++ program, then each segment will often correspond to an object.

Section 5. then describes how the enhanced TOP-C model maps onto a DSM architecture. In particular, there is an important issue of how the multiple segments of the TOP-C environment map onto the multiple pages of a DSM system. We are still in the process of obtaining a suitable DSM, and so we have not had the opportunity to test TOP-C in this environment. Nevertheless, a paper analysis describes many of the DSM features that we expect will be necessary for TOP-C to run efficiently on top of DSM.

2. The TOP-C Model

The TOP-C model has been described in [7]. The model is sufficiently flexible to also be easily ported to interactive languages [5, 6]. The model has also been applied to metacomputing [9], due to the ease of checkpointing the current state and sending a copy of that state to a new process joining the computation. The model has been successfully used in a variety of applications [4, 8, 10, 11, 12, 17].

The model allows a single file of application code to be executed as a sequential, SMP, or distributed memory application, by simply linking with a different library. Portability is emphasized by building on top of a POSIX threads library (for SMP) or MPI [14] (for distributed memory). MPI was chosen as a widely available message-passing standard, with good efficiency. The TOP-C distribution also contains its own small, unoptimized subset implementation of MPI, allowing one to quickly set up a small, self-contained application. Further, the portability of TOP-C makes it easy to re-target to another message-passing platform, such as PVM. TOP-C is freely distributed at `ftp://ftp.ccs.neu.edu/pub/people/gene/top-c/`.

The programming style is SPMD (Single Program, Multiple Data). This is executed in the context of a master-slave architecture and an environment or global state. This environment receives lazy, incremental updates, in a fashion that will be made clear later.

The user interface has purposely been kept simple by restricting the user interface to a single, primary system call: `master_slave()`. That function requires as parameters, four application functions declared by the user: `set_task_input()`, `do_task()`, `get_task_output()` and `update_environment()`. The philosophy is to present the higher-level task abstraction to the application. This should be contrasted to lower level interfaces that present either a message-passing abstraction or a shared memory abstraction.

The *task* is the first abstraction. The first two application-defined functions, `set_task_input()` and `do_task()`, implicitly define the input-output behavior of the task. The third function, `get_task_output()`, returns an *action* to be taken, based upon the task output. The three primary actions are `NO_ACTION`, `REDO`, and `UPDATE`. When the application specifies the `UPDATE` action, the application-specific function, `update_environment()` is called on each process (including the master). The routine, `update_environment()` uses the task output to introduce an incremental update.

The figure below illustrates the flow of control between master and each of several slaves for a task.

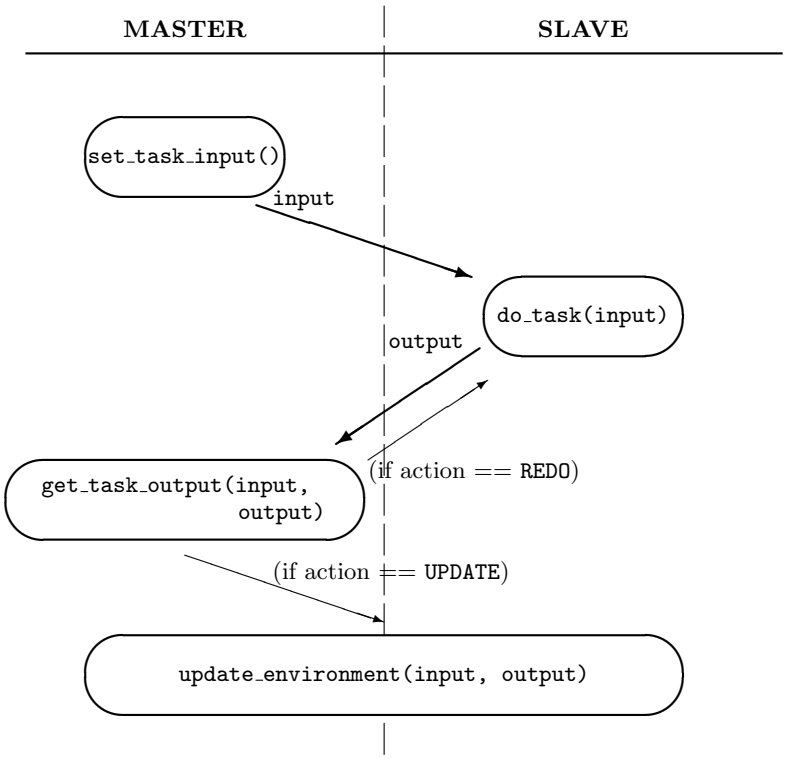

Figure 2.1. TOP-C Programmer's Model

A process always completes its current operation, before reading a pending message for the next operation. A message from the master to a slave requesting an update to the slave's copy of the environment always takes precedence over a message specifying a new task. A `REDO` action results in

the original task input being sent back to the same slave, typically after a message to update the environment.

In addition to the task, the second key to the TOP-C model is the *environment* (global state). The environment, like the task, is not explicitly declared by the application. Rather, it is implicitly defined by the application routines. Each of the four application routines may read the most recent local environment. However, only `update_environment()` may modify the data in the environment. The environment is read and written only by the application routines, and not by any TOP-C system routine.

The most important issue for TOP-C is to allow tasks to concurrently read and make a request to modify the environment. As seen in figure 2., a decision to modify the environment can only happen if `get_task_output()` returns an `UPDATE` action. This action both allows TOP-C to record at what "time" the environment was last modified, and to then call `update_environment()`. In the case of distributed memory, `update_environment()` is called on each process, including the master. In the case of shared memory or sequential code, `update_environment()` is called only on the master.

3. Concurrency Issues for Shared Memory

Note that for any shared memory system (not just TOP-C), there is an inherent reader-writer problem when one thread (in this case the master) writes to a region of memory while another thread is reading the same region of memory. The TOP-C methodology reduces this to a single writer-multiple reader problem. The TOP-C solution is to allow both memory operations to proceed, but to later detect the memory collision and account for it. The method is analogous to the method of "optimistic concurrency" in distributed databases.

Concurrency is maintained in TOP-C in an application-specific manner. The system provides a utility, `is_up_to_date()`, callable from within the application routine, `update_environment()`. This routine will determine whether the environment was modified on the master after the task input under consideration was generated on the master, and before the task output was received by the master. Any memory collisions are a special case of this more general situation, and so will also be detected.

If the environment was not modified, then the application trivially attains perfect concurrency. If the environment was modified, then the application routine, `get_task_output()`, may either return a `REDO` action, or employ an application-specific technique to "patch" the task output to take account of the modified environment. The `get_task_output()` routine receives the task input, in addition to the task output, precisely to make it easier to patch the output.

The effect of this concurrency strategy is that the environment acts as a single large "page" of memory. If any task causes the page to be "touched",

then all processes may have to read an update to the page. The page update is handled in a lazy manner, providing a type of latency hiding. However, the presence of only a single, atomic environment effectively means that false sharing of data is widespread within the system. This is the current state of TOP-C.

The issue of false sharing of a single monolithic environment tends to especially hurt TOP-C applications that require a shared memory model. This occurs because of a natural dichotomy in TOP-C applications. Applications that require only a smaller amount of memory for the environment tend to run comfortably in the distributed memory model, in which the environment is replicated among many processes. However, applications requiring a large amount of memory for the environment will prefer a shared memory environment. Otherwise, the cost of physical memory often makes it uneconomic to find a site with sufficient memory on each processor to allow the replication of a large environment within each process.

Thus, large environments favor a shared memory model. This software view of memory can be achieved either by an SMP architecture or by a DSM architecture on top of many workstations. The next section discusses an experimental version of TOP-C that better accommodates a shared view of memory by providing multiple pages, or segments, within the environment.

4. Multiple Segments within an Environment

In the experimental TOP-C model, the environment is replaced by multiple segments. The use of multiple segments forces us to change one command and one action in the TOP-C model: `is_up_to_date()` and `UPDATE`. All other aspects of the TOP-C retain the same simplicity.

Recall that the TOP-C environment is never explicitly declared. Rather it is implicitly defined by the application programmer as those portions of memory within a slave process that are read by `do_task()` and that are read or written to by `update_environment()`. (In addition, the master routines `set_task_input()` and `get_task_output()` may also read the environment.)

In our implementation of segments, we retain this idea that segments are implicit referenced, but never explicitly declared. Since the environment is replaced by segments, the utility `is_up_to_date()` must be extended to include a single parameter, specifying for which segments the query is being made. Currently, this parameter is specified as a string representing a set of numbers. For example, `"1,3,5-7"` represents segments 1, 3, and 5 through 7.

Second, the command `update_environment()` is now used to update one or more segments. It would be possible to add an additional requirement for the application programmer to have `update_environment()` return a string, such as `"4-8"`, indicating which segments are being updates. This would allow TOP-C to maintain an internal table that updates a timestamp

for each segment, and then answer any application queries of the form `is_up_to_date("1,3,5-7")`. However, it was felt to be a simpler syntax to instead extend the `UPDATE` action returned by `get_task_output()`. Since the application programmer already must return the action `UPDATE` (implemented as a C constant), we now require the application programmer to instead return a parametrized action such as `UPDATE("4-8")` (implemented as a C function macro).

It is clear that the internal table of timestamps for each segment can be maintained only on the master process, since queries of the form `is_up_to_date()` and updates of the form `UPDATE()` both originate on the master process. As each new task originates on the master, a new task ID is issued as a monotonically increasing sequence. The timestamps for each segment are then implemented as task ID's.

So an `is_up_to_date()` query can be answered by TOP-C simply by determining the task ID of the current task being processed by `get_task_output()`. That current task ID is compared with the maximum of the timestamps for each segment being queried by `is_up_to_date()`. Those timestamps are maintained by TOP-C in its internal table, and are task ID's corresponding to the last `update_environment()` for each queried segment. If the current task ID is "newer" (larger), then TOP-C returns true. Otherwise, it returns false.

Thus, the extensions to `is_up_to_date()` and `UPDATE()` impose a minimal additional burden on the TOP-C application programmer, while providing strong benefits in the form of higher concurrency. The partition of the environment memory into segments by the application will often be a natural extension of the application. For example, large application tables or other arrays can be subdivided by partitioning the index set into equal subintervals. Object-oriented applications will often partition their environment by associating an object ID with each object, and associating a TOP-C segment with the memory used by an object. The object ID can then also be used as a segment number.

5. TOP-C over Distributed Shared Memory

Existing DSM systems primarily provide physical memory management and memory consistency. TOP-C provides memory management in the form of implicitly specified TOP-C segments, where the user is responsible for the memory organization, and the TOP-C framework provides consistency management for this memory. Therefore the functionality of TOP-C and a DSM system intersect in the area of memory management. This section discusses the possible benefits and design of a combined system. There is not yet an implementation of the ideas in this section.

The introduction of shared memory to TOP-C introduces a new problem that was not present in the distributed memory of TOP-C. When the master

calls `update_environment()`, writes on the master take effect immediately on the slave, due to the shared memory. This is handled in SMP through a standard single-writer–multiple-reader solution by which readers may later re-read any modified segment through a `REDO` action. Nevertheless, this strategy also imposes a burden on the application writer in that `do_task()` may return a wrong answer after reading inconsistent data, but it must be guaranteed never to hang due to inconsistent data. DSM systems can emulate the lazy updates of TOP-C under distributed memory by implementing lazy release consistency.

Many DSM systems, such as TreadMarks [1], Quarks [16] and the earlier Munin [2] system, support release consistency. *Release consistency* allows for a weaker memory model in which an *acquire* operation is required before reading or writing a shared variable, and a *release* operation is required before another processor can acquire a shared variable. Release consistency allows initiation of a new acquire operation without waiting for pending reads to complete, and it allows a new write without waiting for pending release operations to complete.

A typical implementation of release consistency is to implement two library routines, acquire and release, (`Tmk_lock_acquire(lock_handle)` and `Tmk_lock_release(lock_handle)` in the case of TreadMarks), which operate on a lock handle (an integer in the case of TreadMarks). After an acquire operation, all writes by the application are noted by the DSM system until a corresponding release operation. (Interception of writes can be implemented by the UNIX system call, `mprotect()`.) If a second process acquires the same lock, then all of the modified pages will be replicated on the second process.

Release consistency is typically implemented in one of two variations. These two variations differ in how to handle write updates. The first variation is lazy release consistency. In lazy release consistency, a write update occurs only *after* the call to `release()` by the writing process, and when a second process then calls `acquire()` in an attempt to access the same page of memory. The second variation is eager release consistency. In this variation, modified pages are updated for all processes holding a copy of the page at the time of the call to `release()`. This update can be "batched" for efficiency, but the original call to `release()` may not be seen to complete by a second process until the second process has received the "eager" write updates.

The preferred DSM policy for TOP-C is one of *lazy release consistency* in which there are no page updates seen by other processes and no page invalidations until after the call to `release()` and at the time of a second call to `acquire()`. This mimics the TOP-C memory model of lazy, incremental updates. This fits well with the TOP-C methodology, in which writes to any one TOP-C segment are likely to be infrequent.

If TOP-C were implemented on top of a DSM system, this would require appropriate calls of `acquire()` and `release()` by TOP-C to the

underlying DSM system. One would call `acquire()` before a call to `update_environment()` and `release()` after the call. Before a call to `do_task()` (on a slave), one would call `acquire()` immediately followed by `release()` in order to receive the modified pages.

If one has implemented multiple segments of the environment in TOP-C, one would invoke a different lock handle for each segment. It might become necessary for `update_environment()` to take an additional argument, specifying which segment to update. TOP-C would then guarantee to call `update_environment()` repeatedly, once for each segment that needs to be updated.

Plans are underway to test TOP-C on top of a DSM system. The experimental version of TOP-C (using shared memory) will be tested. This will provide important feedback about merging the TOP-C shared memory model with the shared memory model used by DSM.

1. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, Vol. 29, No. 2, pp. 18-28, February 1996.
2. J. Carter, J. Bennett, and W. Zwanpoel, Implementation and Performance of Munin, *Proc. 13th ACM Symp. Operating System Principles*, 1991, pp. 152–164.
3. R. Chow and T. Johnson, *Distributed Operating Systems and Algorithms*, Addison Wesley Longman, 1997.
4. G. Cooperman, "Practical Task-Oriented Parallelism for Gaussian Elimination in Distributed Memory", *Linear Algebra and its Applications* **275-276**, 1998, pp. 107–120.
5. G. Cooperman, GAP/MPI: Facilitating Parallelism, *Proc. of DIMACS Workshop on Groups and Computation II* **28**, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, L. Finkelstein and W.M. Kantor (eds.), AMS, Providence, RI, 1997, 69–84.
6. G. Cooperman, STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems, *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, ACM Press, 126–132.
7. G. Cooperman, TOP-C: A Task-Oriented Parallel C Interface, 5^{th} *International Symposium on High Performance Distributed Computing* (HPDC-5), 1996, IEEE Press, 141–150 (software at `ftp://ftp.ccs.neu.edu /pub/people/gene/top-c/`).
8. G. Cooperman, L.Finkelstein, M.Tselman and B.York, Constructing Permutation Representations for Matrix Groups, *J. Symbolic Computation* **24**, 1997, pp. 1–18.
9. G. Cooperman and V. Grinberg, "TOP-WEB: Task-Oriented Metacomputing on the WEB", *International Journal of Parallel and Distributed Systems and Networks* **1**, 1998, pp. 184–192; a shorter version appears as: "TOP-WEB: Task-Oriented Metacomputing on the Web", G. Cooperman and V. Grinberg, *Proceedings of Ninth IASTED International Conference on Parallel and Distributed Computing and Systems* (PDCS-97), IASTED/Acta Press, Anaheim, 1997, pp. 279–286.

10. G. Cooperman and G. Havas, Practical parallel coset enumeration, *Proc. of Workshop on High Performance Computation and Gigabit Local Area Networks*, G. Cooperman, G. Michler and H. Vinck (eds.), Lecture notes in control and information sciences **226**, Springer Verlag, pp. 15–27.
11. G. Cooperman, G. Hiss, K. Lux, and Jürgen Müller, The Brauer tree of the principal 19-block of the sporadic simple Thompson group, *J. of Experimental Mathematics* **6**(4), 1997, pp. 293–300.
12. G. Cooperman and M. Tselman, New Sequential and Parallel Algorithms for Generating High Dimension Hecke Algebras using the Condensation Technique, *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, ACM Press, 155–160.
13. G.C. Fox, W. Furmanski, M. Chen, C. Rebbi and J. Cowie, WebWork: Integrated Programming Environment Tools for National and Grand Challenges, *Proc, of Supercomputing '95*.
14. W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1994.
15. J. Protić, M. Tomašević, V. Milutinović, Distributed Shared Memory: Concepts and Systems, IEEE Computer Society Press, 1998.
16. M. Swanson, L. Stoller, J. Carter, "Making Distributed Shared Memory Simple, Yet Efficient", *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'98)*, pages 2–13, March, 1998.
17. M. Tselman, Computing permutation representations for matrix groups in a distributed environment, *Proc. of DIMACS Workshop on Groups and Computation II* **28**, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, L. Finkelstein and W.M. Kantor (eds.), AMS, Providence, RI, 1997, 371–382.