TOP-C: A Task-Oriented Parallel C Interface

Gene Cooperman[1,2]
College of Computer Science
Northeastern University
Boston, MA 02115
gene@ccs.neu.edu

Abstract

The goal of this work is to simplify parallel applications development, and thus ease the learning barriers faced by non-experts. It is especially useful where there is little data-parallelism to be recognized by a compiler. The applications programmer need learn the intricacies of only one primary subroutine in order to get the full benefits of the parallel interface. The applications programmer defines a high level concept, the task, *that depends only on his application, and not on any particular parallel library. The task is defined by its three phases: (a) the task input, (b) sequential code to execute the task, and (c) any modifications of global variables that occur as a result of the task. In particular, side effects (which change global variable values) must not occur in phase (b). Forcing the user to re-organize his computation in these terms allows us to present the applications programmer with a single global environment visible to all processors (whether on a SMP or a NOW architecture), in the context of a master-slave architecture.*

Both a shared memory implementation (running on an SGI or SUN Solaris architecture) and a NOW memory implementation (running on top of MPI) are described. The implementations were tested by a naive program for integer factorization, and by a more sophisticated Todd-Coxeter coset enumeration. Integer factorization was chosen so as to exercise the major features of TOP-C in an unambiguous context.

[1]Partially supported by NSF Grant CCR-9509783.
[2]Appeared in HPDC-5, 1996, IEEE Press, pp. 141–150

1 Introduction

TOP-C is a task-oriented parallel C interface. It presents a master-slave task architecture that greatly eases the parallelization of code. It is intended for applications where a compiler would have difficulty recognizing opportunities for data-parallelism.

The model has been implemented for both shared memory processors (SMP) and networks of workstations (NOW). There is also a sequential version useful during development, which runs the same application code. Ease-of-use has been a strong motivation behind its design. For this reason, TOP-C is organized in a SPMD style, with one primary subroutine call to invoke it. Its main features are: (a) task-parallelism, (b) a single shared, global data structure, and (c) restricted master-slave communication. Further, there is a data structure shared among all processors, whose modification is restricted by a certain protocol. This allows the system to notify a task if its computation was done with respect to a shared data structure that has later been modified. By presenting this information directly to the application programmer, an application-dependent decision can be made as to whether the concurrency was valid or whether a portion of the computation should be re-computed. TOP-C has been used to develop a parallel coset enumerator on up to 16 processors of a SGI Power Challenger Array with nearly linear scalability [6] (also see section 7.1). It uses the Todd-Coxeter algorithm for a mathematical group presented by relations, in an algorithm that is reminiscent of Knuth-Bendix or Gröbner bases.

The master-slave model has particular advantages to help non-experts over the learning barrier. It is deadlock-free. Debugging and reasoning about program logic are aided, since the master can provide a trace of all messages. This implies that the user is presented with a total event ordering for initiation and completion of tasks. Further, that total ordering is consistent with the partial event ordering for task initiation and completion by the many concurrent processes.

The system is described in outline in section 2. Its details are described through an example in section 3. Section 4 contains a discussion of the issues of such a model. Section 5 discusses tracing and debugging, followed by a description of the implementation (section 6) and some timings on three parallel/distributed architectures (section 7).

1.1 Previous work

TOP-C grew out of efforts to parallelize problems in symbolic and especially algebraic computation. The experience in symbolic computation presents a sharp contrast to the experience in numeric computation, where there was much early work using data-parallelism. Symbolic computations tend to have many fewer opportunities for data parallelism. Consider for example the Euclidean GCD algorithm (greatest common divisor) and its many generalizations to polynomial rings and other algebraic domains. This lack of data-parallelism may be partially responsible for the lower level of parallelism in the symbolic algebra user community. Although there was much early work in parallelism for symbolic algebra [8, 16, 22], user practice continues to be dominated by such sequential systems as Axiom, MAPLE, Mathematica and Reduce. This observation provided a strong motivation for keeping the TOP-C parallel interface as simple as possible.

DSC [9] provides an early approach from symbolic algebra with a philosophy like that of TOP-C. DSC runs in LISP and C, and has been used for primality testing and polynomial factorization over large finite fields. It employs a RPC model, although it can also distribute C or LISP code to the remote processor for execution.

A forerunner of TOP-C is STAR/MPI [4], which is useful primarily for coarse-grained parallelism. STAR/MPI runs only in interpreted languages using MPI [13, 14] over a network of workstations. (It uses primarily the point-to-point communication layer and could easily be ported to another library capable of distributed memory, such as PVM [15].) STAR/MPI runs on top of GCL LISP and GAP (a general purpose language for “Groups And Programming”) [18]. It is implemented through a layer that sends a LISP or GAP expression as a string for parsing and evaluation on the remote processor. It does not run on shared memory machines due to the difficulty of retro-fitting a parallel garbage collector.

To date, STAR/MPI has been used for finding a permutation representation of degree 9,606,125 (acting on conjugacy classes) from a matrix representation for the sporadic simple group Ly. The original computation required 4-1/2 days on a SPARC-10, and allowed the permutation generators to be stored and distributed to other researchers for the first time [5]. The parallel version (using ten 25 MHz SPARC-10's) reduced the time from 4-1/2 days to 1/2 day [20].

The system has also been used with a new algorithm for the condensation method in order to obtain for the first time a Hecke algebra (condensation algebra) with matrix dimension 5693 over the finite field GF(2), for the sporadic simple group J_4, which has a permutation representation of degree 173,067,389 [7]. The computation required four 75 MHz SPARC-5's and two 25 MHz SPARC-10's running over 3 days. The system has also been successfully used in brief experiments parallelizing pre-existing sequential code for the Macsyma implementation (on top of GCL Common LISP) of Gröbner bases by Zacharias [21] and for the GAP implementation of the Schreier-Sims test for permutation group membership (implemented jointly with A. Hulpke of RWTH, Aachen, Germany).

Hulpke has also used this system to parallelize polynomial factorization modulo different primes (using Hensel lifting) and to identify Galois groups by computation of resolvent polynomials in parallel.

2 Basic Concepts

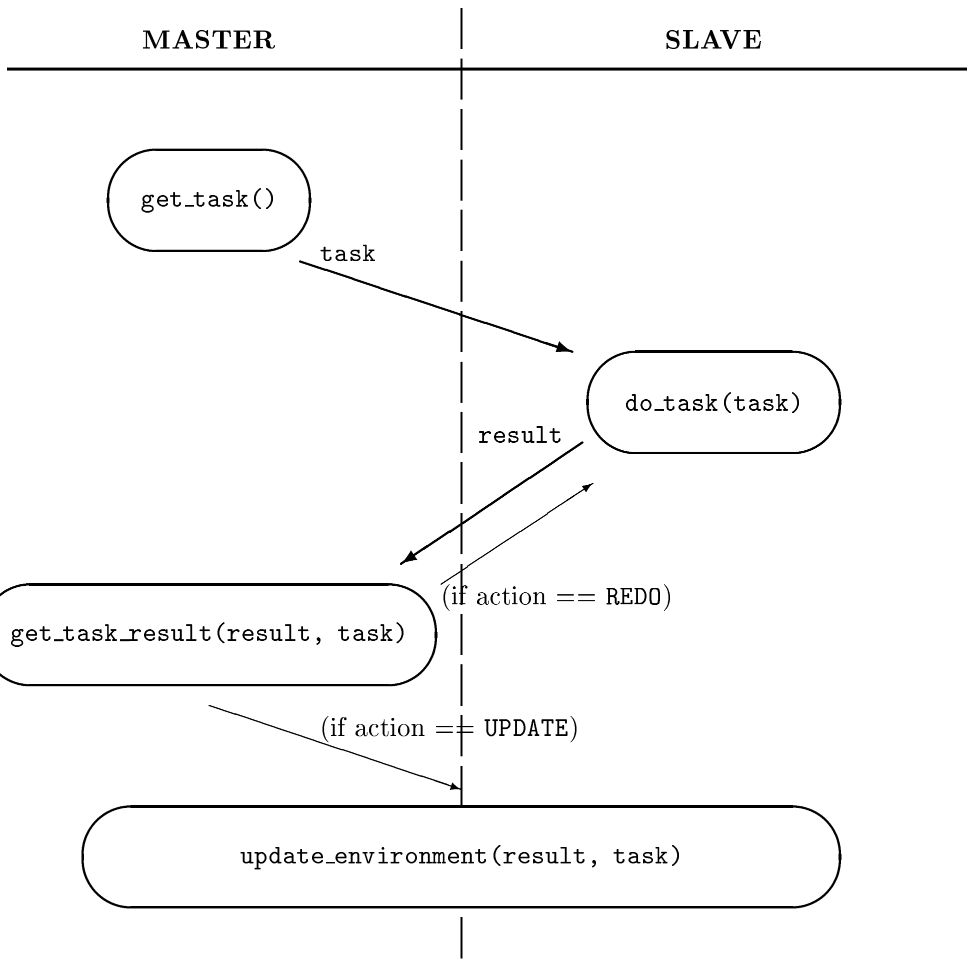
A *master-slave architecture* consists of a unique processor, designated the *master*, and arbitrarily many other processors, designated *slaves*. Communication is constrained to pass only between master and slaves, and not among slaves. (An exception is described in section 3.4.) As we shall see, communication concerning a particular task is further constrained, in that the first communication concerning a new task must be initiated by the master. Hence, the master generates new tasks to be done, and the slave executes them.

The primary interface to TOP-C is an invocation of the form:

```
master_slave( get_task, do_task,
  get_task_result, update_environment );
```

where the four variables, `get_task`, `do_task`, `get_task_result` and `update_environment`, are pointers to functions defined by the user. This call is embedded in a user program that is executed on all processors, thus maintaining a SPMD (Single Program, Multiple Data) programming style. This invocation causes a parallel program to execute. In the special case of a single slave, the program is equivalent to the sequential pseudo-code, below. The life cycle of a single task is displayed graphically on the next page. In the parallel version, different tasks may be overlapped arbitrarily.

The values of the variables `task` and `result` can be arbitrary, and are defined only by the return values of the user functions `get_task()` and `do_task()`. In elementary applications of `master_slave()`, the action will always be



```
while ( NOTASK <> (task = get_task()) ) do                   [on master]
  redo:
  result = do_task( task );                                   [on slave]
  action = get_task_result( result, task );                  [on master]
  switch ( action )
    case NO_ACTION: /* do nothing */;
    case UPDATE:  update_environment(result, task); [on master and slave]
    case REDO:    goto redo;
    case CONTINUATION:  ;                [action defined in section 3.4]
```

NO_ACTION. The reader may wish to consider only this special case on a first reading (in which case the function update_environment() is not used).

Of the four user-defined functions, the routine get_task() executes on the master, the routine do_task() on a slave, the routine get_task_result() on the master, and the routine update_environment() on the master and each slave. The system arranges to execute multiple tasks (one on each slave) at the same time.

In addition to the four user-defined functions, the fundamental concepts of TOP-C are the *task*, the *result*, the *action*, and the *environment*. A *task* consists of a task input or *task description*, a routine, do_task() for executing the task, and resulting modifications to be applied to a global data structure shared among all processors, the environment. A *task description* is a user-defined data structure that is the input to a routine, do_task(). The routine, do_task(), is called with a single argument, the current task description. The routine may also read values from the environment. If a program in TOP-C is to be efficient, then the large majority of the CPU time of the algorithm should be spent in calls to do_task(). This implies, in particular, that most calls to do_task() should not require time-consuming updates to the environment.

The return value of do_task() is the *result*. This and the original task description are the input parameters for get_task_result(). This latter routine returns an *action* that controls further options for processing of the task as depicted in the figures. The four possible actions that can be returned are NO_ACTION, UPDATE, REDO, and CONTINUATION(param), where param is a user-defined data structure. These options are described in further detail in the later sections.

The *environment* is a set of variables, along with their values, shared among all processors. More precisely, the environment is the set of variables whose values are modified by the routine update_environment(). The SPMD style of programming allows the environment to be shared among all processors while requiring no data declarations by the user. The environment can usefully include both global variables and local variables that are in the lexical scope of update_environment() and one or more of get_task(), do_task() and get_task_result(). The functions get_task(), do_task() and get_task_result() may read values from the environment, but only update_environment() should be allowed to set values in the environment.

3 Example

The usage of TOP-C is illustrated by an extended example for naive parallel integer factorization. For clarity of exposition, we ignore many possible optimizations, such as stopping the sieve after testing the square root, and we especially ignore the existence of more sophisticated factorization algorithms. The reader who has absorbed the lessons of this section will have no trouble applying them in a more sophisticated manner.

3.1 A naive, parallel primality test

The following code employs a variation of the sieve of Eratosthenes, in which previous factors are not saved. This version uses only the trivial parallelism, but is useful for illustrating the parallel notation. Note that the global variable num_to_factor is shared between get_task() and get_task_result().

```
static int num_to_factor, last_num;

int IsPrime( num )
{ last_num = 1;
  num_to_factor = num;
  master_slave( get_task, do_task,
    get_task_result, update_environment );
  if ( num_to_factor == 1) return 0 /* false */;
  else return 1 /* true */; }

void *get_task()
{ last_num = last_num + 1;
  if ( last_num > num_to_factor )
    return NOTASK;
  return (void *)last_num; }

void *do_task( num )
     int num;
{ return (void *)(num_to_factor % num == 0);}

int get_task_result( result, num )
     int result, num;
{ if ( result == 1 /* true */ )
    num_to_factor = 1;
  return NO_ACTION; }
```

3.2 Parallel integer factorization

This section introduces several additional features of TOP-C. This time, the routine update_environment() is used to maintain uniform values of the variable num_to_factor across all processors as part of the global environment. The same is done for the factors array in this example, although the latter need not be global.

```
static int num_to_factor, last_num,
           factors[1000];

int *Factor( num )
     int num;
{ factors[0]=0;
  last_num = 1;
  num_to_factor = num;
  master_slave( get_task, do_task,
    get_task_result, update_environment );
  return factors; }

void *get_task()
{ last_num = last_num + 1;
  if ( last_num > num_to_factor )
    return NOTASK;
  return (void *)last_num; }

void *do_task( num )
     int num;
{ return (void *)(num_to_factor % num == 0);}

int get_task_result( result, num )
     int result, num;
{ if (result == 0/* false */)return NO_ACTION;
  if ( ! is_up_to_date() ) return REDO;/*(*)*/
  return UPDATE; }

void update_environment( result, num )
     int result, num;
{ while ( num_to_factor % num == 0 )
    { factors[ ++factors[0] ] = num;
      num_to_factor =
        num_to_factor / num; } }
```

Finding all the factors requires updating an environment. Otherwise, a slave would note that both 2 and 4 divide 12, and there is a danger of storing both factors. Hence, we call update_environment() in a manner that is clear from section 2.

The function is_up_to_date() returns false (boolean 0) if and only if update_environment() has been called by the master during the interval between the time when the master invoked get_task() to generate the "task" argument (num) of do_result() and the time when get_task_result() was invoked with the same task num. In the case at hand, if num_to_factor has changed since the task num was last sent to do_result(), then is_up_to_date() is guaranteed to return false.

The starred statement from get_task_result() should be given special notice:

```
  if ( ! is_up_to_date() ) return REDO;/*(*)*/
```

It is possible to avoid the use of is_up_to_date(). The master could re-test if num is still a factor of num_to_factor. The use of is_up_to_date() is a more efficient alternative. If num was previously tested and found not to be a factor, then that will continue to be the case with respect to the modified num_to_factor, and so NO_ACTION can be immediately returned. On the other hand if, for example, 2 was found found to be a factor of 12, and a concurrent task found 4 to be a factor, then the second concurrent task will be re-executed. The combination of is_up_to_date() and REDO is a standard idiom in programming master_slave(). A REDO action guarantees to send the original task back to the original slave. This feature can be used to potentially make do_task more efficient by locally caching previously computed data.

3.3 A subtle bug

There is still one bug in the above code. One of the factors returned might be a composite number and not a prime. If Factor(12) is invoked, causing three slaves to examine in parallel the three factors, 2, 3, and 4, then the slave examining the factor 4 might return first. In this case, Factor(12) would return array values {4, 3}. The following modifications fix this bug.

```
/* Change line of get_task_result
   commented by "(*)":  */
  if ( ! is_up_to_date() &&
       num > Maximum(factors) )
    return REDO;  /* (*) */

/* Add to beg. of update_environment(): */
        if ( num < Maximum(factors) )
          RemoveMultiples( num );
```

In an efficient version, one would of course add a line at the end of update_environment() to cache and update the latest value of Maximum(factors) in a static, local variable. The function RemoveMultiples(num) is defined to remove all elements, x, of the array factors that are multiples of num. It also updates num_to_factor by multiplying it by each x. Thus, composite factors are eventually caught and corrected.

Note an interesting phenomenon. Consider again the example, Factor(12). In the modified code, if the slave examining the factor 2 were slow to return, then num_to_factor might take on values, 12, 4, 1, 2. For this reason, the system will always call get_task() one last time after all slaves have returned, to determine if the last slave has altered the interim determination that the job is done.

3.4 Other features

There is one other action not previously discussed. The routine get_task_result() can return CONTINUATION(next), where next is a second input. That parameter is returned to the same slave that generated the task. This feature allows extensions to the basic paradigm. It provides full generality in communication patterns. For example, a slave can send a question to the master in the middle of his task. If master and slave follow a common protocol, then the master can provide a reply to the slave via the parameter of the CONTINUATION() action. One can even design a protocol by which several slaves communicate with each other, using the master a "postmaster". A utility, get_last_source, is provided for this and other protocols. It allows the master to determine a unique id for the last slave that communicated.

For efficiency reasons, one may also wish to invoke is_master(), which returns true if and only if invoked on the master. This allows one to compute data structures (or read them from a file) only on the master if they are particularly expensive of space or time. Of course, such private data structures will not be part of the shared environment.

There are other features not discussed here that can further extend the generality of the TOP-C programming model. For shared memory machines, a utility, ms_barrier(), is offered. This is useful in the context of an update_environment(), since a slave might be accessing data at the same time that update_environment() is causing it to be modified. Some systems, such as Indigo [17], provide the applications programmer with finer control over sharing of memory, and this could provide a more efficient alternative to ms_barrier().

The generality of the system can be further augmented by allowing arbitrary messages directly between slaves. This eliminates the inefficiency of the "postmaster" protocol referred to above. However, such a drastic departure from the original paradigm is not recommended, since it risks losing many of the benefits, such as the absence of deadlock and the total ordering of events through communication via the master.

4 Features of the TOP-C model of parallelism

The methodology described here was chosen for its simplicity, so as to lower the learning barrier for people in symbolic algebra wishing to write parallel code. The TOP-C model fits many problems in symbolic algebra, which are *weakly inter-dependent.* This means that the environment needs to be modified only infrequently. Such a property often achieves nearly linear scalability. The proposed model encourages this property, since modifying the environment requires the user invoke an additional routine, update_environment(), through the UPDATE action. This principle can be codified into a figure of merit, Φ, for weak inter-dependence. Let T be the total number of tasks executed, N the number of processes, and M the total number of times the global data structure needs to be modified. Then

$$\Phi = T/(M * N).$$

Hence, trivially parallel programs (no task interaction) will have an infinite value for Φ.

The model is especially economical in its use of available communication bandwidth. This is important for LAN's and shared memory environments. An Ethernet-style LAN supports only one packet at a time (assuming no subnets). On a shared memory environment, memory contention can restrict simultaneous communication by multiple processors.

Applications using TOP-C tend to have small and infrequent messages that are distributed without sharp bandwidth peaks. The SPMD programming style encourages redundant computation of large initial data structures on each processor in parallel, instead of a single computation along with large messages to propagate the result. Further, communication and computation phases on distinct slaves will tend to overlap as slaves become out of phase with each other.

Total communication requirements can be higher after an UPDATE action, when large simultaneous messages might be sent to all slaves. However, broadcast messages can be optimized for lower overhead. In the case of Ethernet, this implies broadcast packets. In the case of shared memory, a single shared copy can be written to memory. Similar strategies are available on other networks.

The TOP-C model shares some of the features of other models. As with the RPC model, TOP-C is deadlock-free. As with the distributed shared memory (DSM) model, a user can run the same code, unchanged, on both distributed and shared memory architectures. It also has a single shared environment, in common with the DSM model. Nevertheless, TOP-C is distinguished from DSM in that the user interrogate the system (via `is_up_to_date()`) whether the latest shared environment is available, and he/she chooses whether to wait for the lastest update of the environment (via a `REDO` action) or whether to make use of the currently available environment (possibly via an `UPDATE` action).

The task-parallel model presents a cousin of this model. For the sake of concreteness, we consider CC++ [2] (Compositional C++) and Fortran M [1, 11] (along with the Nexus runtime system [12]) as examples of task-parallel environments. The biggest differences are that TOP-C encourages the user to maintain a single, environment (data structure) uniformly across all processors, and communication is more restrictive in TOP-C, as compared to the channels of CC++ or Fortran M.

TOP-C shares both with RPC and with Fortran M and CC++ the property that the primary user concern is to specify a task and define the task input and the task output. This relieves the user from some of the lower level concerns. Heterogeneous processing in TOP-C is not as easy to express as in Fortran M and CC++, but one can easily program heterogeneous tasks. Heterogeneous tasks in TOP-C are programmed by adding a case parameter to the task description that specifies which task is desired. The routine `do_task()` uses this parameter to dispatch to the correct task.

5 Tracing and Debugging

The following strategies have been found to be valuable for code development and debugging. First, one should replace the standard TOP-C library with a provided file, mas-slave-seq.c, which is a generalization of the sequential code in section 2. This reduces the application to a completely sequential program without change to the user's application code, thus allowing the user to quickly distinguish parallel logic errors from sequential logic errors.

The first stage of parallel development is to use one master and one slave. If possible, the master and slave should be the same CPU, so as to minimize network delays and ill effects on other users. When that code works correctly, it can then be tested on two slaves, and finally on all possible slaves.

Another easy testing strategy is to trace all messages to and from the master. One can cause each task description to be printed in the order that it is seen by the master by setting the following variable:

```
master_slave_trace = true;
```

If this produces too much output, or not the right kind of information, one can add print statements both on master or slave. Shared memory implementations allow printing from all processors, and the MPICH implementation of MPI also arranges for output from all processes to be printed on the user console. Of course, output from a slave can appear asynchronously with the master's output, since it must go through the master's processor to reach the user console.

6 Implementation

The implementation has been kept small and simple to enhance portability and maintainability. It requires two layers of code, with the user application constituting a third and highest layer. In the case of shared memory, the lowest layer is a simple implementation of message passing using the SGI, Solaris, or POSIX system calls for threads. This layer contains all computer vendor-specific portions. It consists of 350 lines of code.

In the case of the distributed memory implementation, this lowest layer is an interface to MPI (Message Passing Interface) [13, 14] providing the same services as the shared memory layer. Because the use of MPI is largely restricted to point-to-point communication, it would be easy to replace this with an alternative message-passing implementation. The package described here has been implemented using the MPICH implementation [10]. The distributed memory version of this layer consists of 200 lines of code.

Binary data format is an issue especially for distributed memory in a heterogeneous environment. Although MPI provides some facilities for data storage independent of vendor byte ordering, etc., use of this for C struct and union would add considerable complexity. Currently, one compiles the package with a user-chosen default type (char, int or float), and any type that is not the default must by cast by the user. MPI automatically addresses different binary formats for int and float, but the user is responsible for heterogeneous computing with compound data types, such as struct and union.

The second layer implements the master-slave architecture in a vendor-independent manner, and independently of issues of shared vs. distributed memory. It consists of 450 lines of code.

The highest layer is the user application. The example of the previous sections provides an excellent description of this layer.

7 Timings

7.1 Parallel facilities and experimental methodology

Experiments were carried out on two shared memory and one distributed memory architecture. The first shared memory architecture was Boston University's SGI Power Challenger Array. That computer consists of 18 R8000 CPU's, each running at 90MHz. The CPU's share 2 gigabytes of 8 way interleaved memory. The second shared memory architecture was the Boston University SPARC-1000 with 4 CPU's under Solaris 2.4. The third architecture was a cluster of up to 8 alpha workstations on a fast DEC ring network. All machines were lightly loaded, except for the DEC cluster where machines typically had one other CPU intensive job running. On all architectures, the TOP-C sequential emulator was run using the same factorization code.

A variation of the factorization program in section 3.2 was used for testing. The main difference in the testing version was that a task was re-defined to test the next 10,000 candidate factors, so as to minimize the ratio of communication overhead to task execution time. Two suites of examples were chosen. The first suite consisted entirely of primes, and the program reduced to one of trivial parallelism. The second suite consisted of 10 random integers in the stated range, and average elapsed times for a random input were determined. The random integers were generated once only and the same integers were used on all architectures.

There was also an issue of which numbers to report as relevant. In reporting overall statistics, we chose to consider the total elapsed time.

We also review the results of a more sophisticated application, Todd-Coxeter coset enumeration. The full details are reported in [6]. The Felsch strategy was used, and computations were carried out on the SGI computer. TOP-C was used to parallelize previously developed sequential code by M. Schönert. The group studied was T, which plays an important role in the proof that Engel-4 groups of exponent 5 are locally finite. There are close connections between the Todd-Coxeter algorithm and the Knuth-Bendix algorithm. There were 45,000 tasks performed. The parallel calculation was non-deterministic, but all runs computed a number of cosets that was close to that for the sequential program.

7.2 Results

The number 100,000,007 was checked for being prime. Every number through 100,000,007 was checked as a factor. There was perfect parallelism since no factors were found. The times are all measured by the UNIX shell time command. All times are in seconds. The numbers for the alpha cluster are less reproducible due to variable loadings from other jobs. Nevertheless, it is clear from the table that roughly linear speedup is obtained.

The two columns for the SPARC 1000 represent elapsed (real) time versus total CPU time among all processors. The starred number for the SPARC 1000 is interesting because that machine had only 4 processors. Yet 15 threads ran approximately twice as fast as 4 threads on a lightly loaded machine. Presumably this is accounted for by more opportunities to overlap computation and memory access, leading to fuller bus utilization.

# slaves	SGI	SPARC 1000 (elaps.)	SPARC 1000 (total)	Alpha cluster
SEQ	43	111	111	206
1 slave	57	298	98	252
2 slaves	34	154	100	157
3 slaves	23	99	101	114
4 slaves	18	68	102	89
5 slaves	14	-	-	76
8 slaves	-	-	-	58
10 slaves	7	-	-	-
15 slaves	5	(*)32	112	-

We also chose 10 random numbers in the range 100,000,000 to 100,000,100. Only one case required a significant amount of CPU time ($3 \times 33,333,347$). As one would expect, the time was about 1/3 of the previous time for all architectures. The next largest factorization occurred as $3 \times 5 \times 7 \times 952,381$, and the times were again reduced by a ratio of about $105 = 3 \times 5 \times 7$. In general, there was weak task inter-dependence for typical random numbers and one sees nearly linear speedup.

Finally the results from Todd-Coxeter Enumeration [6] are reported.

`Todd-Coxeter Coset Enum. (Felsch Strategy)`

Enumeration	Time (sec)
1 slave	219
2 slaves	129
3 slaves	99
4 slaves	78
5 slaves	62
10 slaves	38
15 slaves	29

8 Future Work

The current version is freely available in the ftp directory:
`ftp.ccs.neu.edu:/pub/people/gene/top-c/`
A future version is planned that will encompass both shared and distributed memory architectures as part of a single computational model. Since the model presented in this paper uses the same mechanism for both memory models, it should be ideally suited to such an undertaking. Further, by "sharing" the distributed memory at the task level, it should present a higher level alternative to the user-level sharing of distributed memory that is discussed in [17].

9 Acknowledgements

The author gratefully acknowledges Boston University for the use of the SPARC-1000 and SGI Power Challenger Array.

References

[1] K.M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng, "Integrated Support for Task and Data Parallelism", *Intl. J. Supercomputer Applications*, 1994 (to appear).

[2] K.M. Chandy and C. Kesselman, "Compositional C++: Compositional Programming", *Proceedings of the Fourth Workshop on Parallel Programming and Compilers*, Springer Verlag.

[3] G. Cooperman, "GAP/MPI: Facilitating Parallelism", *Proceedings of DIMACS Workshop on Groups and Computation* (held at Rutgers University, June 7–10, 1995), 1996, to appear.

[4] 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, pp. 126–132.

[5] G. Cooperman, L. Finkelstein, M. Tselman, B. York, Constructing Permutation Representations for Matrix Groups, *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '94)*, ACM Press, pp. 134–138.

[6] G. Cooperman and G. Havas, "Practical parallel coset enumeration", preprint, submitted to *Proc. Workshop on High Performance Computing and Gigabit Local Area Networks*, Lecture Notes in Control and Information Sciences, Springer-Verlag.

[7] 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, to appear,

[8] J. Della Dora and J. Fitch, eds., *Computer Algebra and Parallelism*, Academic Press, 1989. (Proc. CAP '88, Grenoble, France, June 1988).

[9] A. Diaz, E. Kaltofen, K. Schmitz and T. Valente, "A System for Distributed Symbolic Computation", *Proc. of Int. Symp. on Symbolic and Algebraic Computation (ISSAC-91)*, ACM Press, 1991, pp. 323–332.

[10] N. Doss, W. Gropp, R. Lusk and A. Skjellum, software at info.mcs.anl.gov in /pub/mpi/mpich-Jul22.tar.gZ, anonymous ftp.

[11] I. Foster and K.M. Chandy, "Fortran M: A Language for Modular Parallel Programming", *J. Parallel and Dist. Comput.*, 1994 (to appear).

[12] I. Foster, C. Kesselman and S. Tuecke, "The Nexus Task-parallel Runtime System", *Proc. 1st Intl Workshop on Parallel Processing*, 1994.

[13] W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1994.

[14] Message Passing Interface Forum (author), "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications* **8**, Number 3/4, 1994.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[16] J.-L. Roch, P. Senechaud, F. Siebert-Roch, and G. Villard, "Computer Algebra on a MIMD Machine", *Proc. Int. Symp. on Symbolic and Algebraic Comp. (ISSAC '88)*, Lecture Notes in Computer Science **358**, Springer Verlag, 1988, pp. 423–439.

[17] P. Kohli, M. Ahamad and K. Schwan, "Indigo: User-level Support for Building Distributed Shared Abstractions", *Proc. of High Performance Distributed Computing '95* (HPDC-4), IEEE, 1995.

[18] M. Schönert et al., GAP – *Groups, Algorithms and Programming* (Manual), Lehrstuhl D für Mathematik, RWTH, Aachen, Germany, 1995.

[19] C.C. Sims, "Computation with Permutation Groups", in *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, edited by S.R. Petrick, ACM Press, New York, 1971, pp. 23–28.

[20] Michael Tselman, "Computing permutation representations for matrix groups in a distributed environment", *Proceedings of DIMACS Workshop on Groups and Computation* (held at Rutgers University, June 7–10, 1995), 1996, to appear.

[21] G. Zacharias, A Groebner basis implementation in LISP, version 202.

[22] R. Zippel, ed., *Computer Algebra and Parallelism*, Lecture Notes in Computer Science **584**, Springer Verlag, 1992, pp. 1–18. (Proc. of CAP'90, Ithaca, NY, June, 1990).