TOP-WEB: Task-Oriented Metacomputing on the Web[1,2]

Gene Cooperman[3] and Victor Grinberg[3]
College of Computer Science
Northeastern University
Boston, MA 02115
gene@ccs.neu.edu and victor@ccs.neu.edu

Abstract

As personal computers on the World Wide Web proliferate, a great source of unused CPU cycles is being unleashed. TOP-WEB employs the task-oriented model of parallelism used by the earlier system, TOP-C [6], which relieves the application programmer of much of the burden of parallelizing an algorithm. A particular strength of TOP-WEB is its support for automatic checkpointing for long-running jobs. This is anticipated as an important requirement for metacomputing on the Web since very large computations may require weeks or months of publicity before accumulating sufficient donations of CPU cycles to make appreciable progress. Other strengths of TOP-WEB are its latency hiding and its small run-time library, suitable for network computers. An additional useful feature of TOP-WEB is its ease of application programming. For example, the computation productively uses additional slave processors that join and depart from the computation at arbitrary times, while requiring no application code to handle such events. Indeed, the application code need make no reference even to how many slaves are present. TOP-WEB also includes a checkpointing facility making computations over many months quite practical. Several other systems [1, 12, 18], of which Charlotte [1] may be best known, have recently been proposed and implemented to also take advantage of these unused CPU cycles. TOP-WEB is distinguished in providing a separate library that can easily be ported to other languages besides C.

1 Introduction

This paper describes a mechanism for implementing distributed CPU cycle servers on the internet. The hope is to take advantage of the vastly increasing amount of free CPU cycles

[1] This paper appeared in *International Journal of Parallel and Distributed Systems and Networks* **1**, 1998, pp. 184-192.

[2] An earlier version of this paper appeared in *Proc. of Ninth Int. Conf. on Parallel and Distributed Computing Systems*, Washington, D.C., 1997.

[3] Partially supported by NSF Grant CCR-9509783.

being made available. The system builds on an existing system, TOP-C [6], that allows dedicated processors within a local network to easily share a computation. TOP-C greatly eases the job of writing parallel software or of converting existing sequential software to parallel. The current system should greatly ease the job of finding free CPU cycles that can be harnessed to a large computation. Here, we use the term "free" in two senses: available cycles, and cycles requiring no monetary payment.

Previously, the best method for finding free CPU cycles might have been at a university, where there were often many student workstations lying idle overnight. In the future, the best method for finding free CPU cycles may be to request donations of free CPU cycles on the Internet from the millions of PC-owners "surfing the net". We now live in a transition period where one need not look too far into the future to see the elements of this vision coming together.

One anticipates that researchers (or other applicants for CPU time — prognostication is a difficult art) will put up a Web page on the Internet that announces a large computation, a popular exposition of the computation and its potential benefits. The same Web page would then provide a "button" on which to click if the viewer of the page would like to contribute some CPU cycles. The act of clicking on the button would then cause client software to be downloaded to the user's machine. The downloaded software would then connect up with the master process and request a task for computation.

Such altruistic models of free computation have several precedents. Consider the donations of free CPU cycles donated to the factorization of RSA129 [12], the GNU philosophy of free software, the many free dictionaries, on-line texts, encyclopedias, and so forth on the network. It is also possible that a mixed model may develop. Just as many free Web pages are supported by advertising, one can imagine providing a service of general intellectual interest in exchange for free CPU cycles. A minimal example would be a live indication of the progress of the computational project.

Several other systems [1, 12, 18] already offer the possibility of attaching CPU cycles offered on the Web. In particular, Charlotte is probably the most developed of these systems. The current system is distinguished by taking a general, task-oriented approach. Two of the systems [12, 18] offer a pure master-slave architecture, but this is considered too restrictive to qualify as a general, task-oriented approach.

The current system is further distinguished by its automated checkpointing of long-running jobs. As explained in section 4, a checkpointing facility is considered absolutely essential for any job that intends to gather truly large numbers of client donors around it. In addition, TOP-WEB emphasizes latency hiding and a small footprint for "thin clients" in a non-trivial, parallel environment.

The remainder of this paper first describes TOP-C (section 2) on which TOP-WEB is based. It is important to understand the TOP-C model in some detail in order to understand its benefits for metacomputing on the Web. That model then provides the background to understand why checkpointing is exceptionally easy to implement in TOP-WEB. This is discussed in section 4. Section 3 describes the lower level architecture and protocols used to set up communication between the master (manager) and the slaves (workers) on the Web. Section 5 discusses how the performance of TOP-WEB compares and differs from TOP-C. Section 6 discusses the issues of a distributed shared memory approach to metacomputing, as exemplified by Charlotte, with a task-oriented approach as exemplified by TOP-WEB.

Finally, Java was considered as the underlying language for TOP-WEB, but it was decided that current implementations of Java are not yet sufficiently efficient as compared to C. These issues and those of legacy code are discussed in section 7.

2 The TOP-C model: Task-Oriented Parallelism

A short summary of the TOP-C model is provided here. For further details, the interested reader should consult reference [6]. The TOP-C software is available at `ftp://ftp.ccs.neu.edu/pub/people/gene/top-c/`. The goal of TOP-C is to provide support for easy functional decomposition into tasks of either a sequential program or a sequential algorithm. The power of TOP-C derives largely from having the application programmer first re-organize his program into application-dependent tasks and then using TOP-C to parallelize at a higher level. It is felt that identification of opportunities for parallelization in an application will typically be application-dependent and the job of a good parallel tool is to provide ways to cleanly pass this issue on to the application. TOP-C, itself, is implemented as a run-time library.

The TOP-C model is sufficiently simple that it has been implemented in much less than 1,000 lines of code. This has been done in two interactive languages, LISP [5] and GAP [4], as well as C. (GAP, or "Groups, Algorithms and Programming", is a general-purpose language for computational group theory.) The model is also sufficiently flexible so that run-time libraries for C have been developed for distributed memory, for shared memory, and for sequential execution within a single process/thread. The distributed version runs on top of MPI (Message-Passing Interface) [13], thus allowing TOP-C to execute on heterogeneous networks of workstations as well as other parallel architectures supported by MPI. The shared memory version of TOP-C runs on top of POSIX threads. A single API is presented to the application code, thus hiding any dependence on code for thread generation or message passing. We have also developed a subset implementation of MPI requiring only 1200 lines of C. This allows us to maintain a small run-time library, which is particularly important for the newly emerging model of "thin clients".

2.1 Previous computational results using TOP-C

The initial testbed domain for TOP-C was computational algebra. The computations in this domain provide the best empirical demonstration that the simplicity of the TOP-C model has not compromised the power of its approach. The simplicity of the TOP-C approach is demonstrated by [3], which demonstrates an efficient, parallel implementation of Gaussian elimination on a network of workstations, while requiring a relatively small modification of a standard sequential implementation of Gaussian elimination. The power of the TOP-C approach is demonstrated by several algebraic computations, each of which was the largest of its kind, to date, and in each case, almost linear speedup with the number of processors was seen. In some cases, as indicated, the computation was done in LISP/MPI (with C subroutines for numerical subroutines) rather than TOP-C. These achievements include: finding a permutation representation of Lyons's group of degree 9,606,125 over $GL(111, 5)$ [7, 17]

(LISP/MPI, 8 SPARC-2's, 12 hours); condensation of a permutation matrix representation of J_4 from permutation dimension 173,067,389 to a matrix representation for an algebra of dimension 5,693 over $GL(112, 2)$ [10] (LISP/MPI, 14 heavily loaded Alpha's, 2-1/2 days); and condensation of a permutation matrix representation of Th from permutation degree 976,841,775 to matrix dimension 1,403 over $GL(248, 2)$ [9] (LISP/MPI, 8 Alpha 300's, one month). A still larger computation is currently in progress using TOP-C. This is the condensation of a permutation matrix representation of J_4 from permutation degree 173,067,389 over $GL(1333, 11)$.

Although the technical details are outside of the scope of this article, one can give an idea of the size of the computation. These computations represent a number of matrix-vector multiplications proportional to the degree. Each matrix-vector multiplication is over $GL(n, p)$ where the dimension of the vector space is n and the field is a finite field of order p. All of the parallelizations were non-trivial ones in which there was significant interaction among results from different slave processes.

Perhaps the most impressive indication of the power of TOP-C is its ability to easily parallelize legacy code. For example, in joint work with Havas [8], we parallelized sequential C code for coset enumeration by Martin Schönert at RWTH-Achen, Germany. That code had been written 6-1/2 years earlier with no thought for parallelization. Approximately 30 lines of a 1400 line C program were modified, and the program was then linked with the TOP-C library to achieve the parallelization. It was later run on a 4-processor shared memory Convex machine. Using this parallel version of Schönert's program, we then enumerated 8,835,156 cosets in 6 hours with linear speedup and thus determined a permutation representation of Lyons's group from a presentation. (A shared memory architecture was chosen in this case due to the large data structure required for the computation.)

2.2 TOP-C programmer's model

The key to the TOP-C [6] programmer's model is to allow the end-user to express his code in terms of high-level tasks reflecting his/her abstract algorithm. The system transparently arranges for tasks to be executed in parallel on different slave processes or slave threads. As long as there is weak dependency between tasks (few conflicting results between parallel tasks), it is possible to achieve good efficiency (and for many applications, a linear speedup with the number of processors). Further, the system provides some support for reconciliation when independent tasks result in conflicting updates to global data structures.

The life cycle of a task can be summarized by either the pseudo-code or the diagram below. The user must provide the four routines, `get)task()`, `do_task()`, `get_task_result()` and `update_environment()`.

A TOP-C or TOP-WEB program is written in a SPMD style as a single program. However, the application programmer is required to write the four routines, `get_task()`, `do_task()`, `get_task_result()` and `update_environment()`. This is typically not a burden since most programs and algorithms have natural, application-dependent task decompositions. Indeed, part of the power of TOP-C derives from allowing the application programmer to first re-organize his program into application-dependent tasks and then using TOP-C to parallelize at a higher level. Each of the four routines is then executed either on master or slave, as indicated by the diagram. The routine, `update_environment()`, is executed on all

```
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:  ;
```

processors, as described below.

If one were to ignore the `update_environment()` routine and the backwards-directed arrow in the diagram, then this would be a typical master-slave model (or manager-worker). The variables, `task` and `result` in the diagram refer to application-defined opaque data structures that are passed from the output of one routine to the input of the next one. It is the two features of an environment and a backwards-directed message that have allowed the model to be adapted to easily parallelize a wide variety of sequential applications.

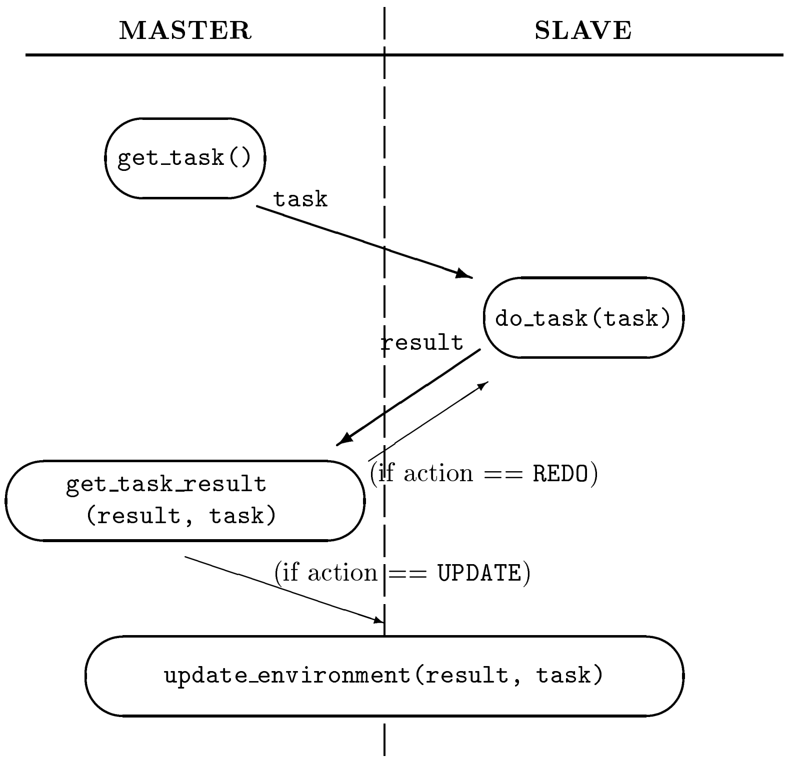
The first distinguishing feature is the *environment*. Conceptually, the environment is a set of global variables, along with their values, shared among all processors. However, the user does not explicitly declare these shared variables. Instead, the user specifies the environment operationally through the additional routine, `update_environment()`. In that routine, the user may modify the values of arbitrary global data. Since TOP-C arranges to eventually call `update_environment()` on all processors, this results in a lazy update to this implicitly declared environment.

The second feature that distinguishes TOP-C from a master-slave model is that it is possible for the application-defined routine, `get_task_result()`, to return a value causing TOP-C to pass a parametrized message back to the original slave process associated with that task. This is important when two tasks are executing in parallel, and they generate independent updates to the environment.

There is a possibility of conflict if two slave processes operating in parallel return results that would result in conflicting updates to the environment. The conflict can be resolved in either of two ways. First, it may sometimes be possible in an application-dependent manner to merge the results of two independent tasks and to implement the routine, `update_environment()`, so as to apply the two updates sequentially and consistently. A trivial example of this occurs when one task requires an update to the environment and the other task does not. This often occurs in applications that sift through many cases in order to apply only a limited number of pertinent cases.

A second, application-independent resolution of conflicting updates is possible through the following definition of `get_task_result()`.

```
int get_task_result( result, task )
     int result, task;
{ if (result == /* no new result to do */)
```



TOP-C Programmer's Model

```
    return NO_ACTION;
  if (! is_up_to_date(task)) return REDO;/*(*)*/
  return UPDATE; }
```

The function, `is_up_to_date()`, returns true if and only if there have been no calls to `update_environment()` on the master process since the task, `task`, was originally generated by `get_task()`. If `get_task_result()` returns `REDO`, then the `task` will be sent back to the original slave on which `result` was produced, and that slave will then re-execute `do_task()`.

In the worst case, every task may require knowledge of the environment as modified by the immediately preceding task. However, even in this case the tasks will be performed sequentially and the application may not have a natural parallelization. However, even in this case, there is an opportunity for some parallelization. Since tasks are "re-done" on the original slave process where they were previously executed, it is possible for that slave process to cache certain partial results that may be independent of how the environment was modified. In this case, such cached results will effectively have been executed in parallel.

3 Software architecture and protocols

TOP-WEB has been implemented in three layers. At the lowest layer, an architecture and protocol have been implemented. This implementation allows a user on the Web to read a web page and then click on a button enabling the user to donate CPU cycles to a large computation. In many respects it is quite similar to that of Charlotte [1] or DAMPP [18]. It differs from these two in that it has been implemented in C rather than Java. This is done largely for reasons of efficiency and legacy code, as described in section 7, but a future version of TOP-WEB is anticipated to be written in Java. This section is concerned primarily with this lowest layer.

The middle layer is MPI. MPI isolates the higher layers from sockets and other communication subsystems, thus providing greater portability. Although there are several good, free MPI implementation, we implemented our own small subset of MPI, sufficient to support TOP-C. The subset implementation was executed so as to maintain a small software "footprint". This is important for "thin clients" and also for background processing that will not disturb the donor of CPU cycles. The MPI layer accounts for about 1200 lines of C code. We thank Markos Kyzas for his assistance in the development of the early stages of the MPI subset.

The MPI subset has had to be enhanced in order to account for new slaves joining the computation and some slaves dying. The emerging MPI-2 [16] standard provides a routine, MPI_SPAWN, as part of the dynamic process management extension, which can be used to generate a new slave process when notified of a new processor by the lowest layer. The MPI standard does not specify what action should be taken when a slave dies. Currently, the MPICH [15] implementation usually dies when a slave dies, while the LAM [11] implementation has facilities for notifying the caller that the slave is dead. Our own subset implementation employs the latter approach.

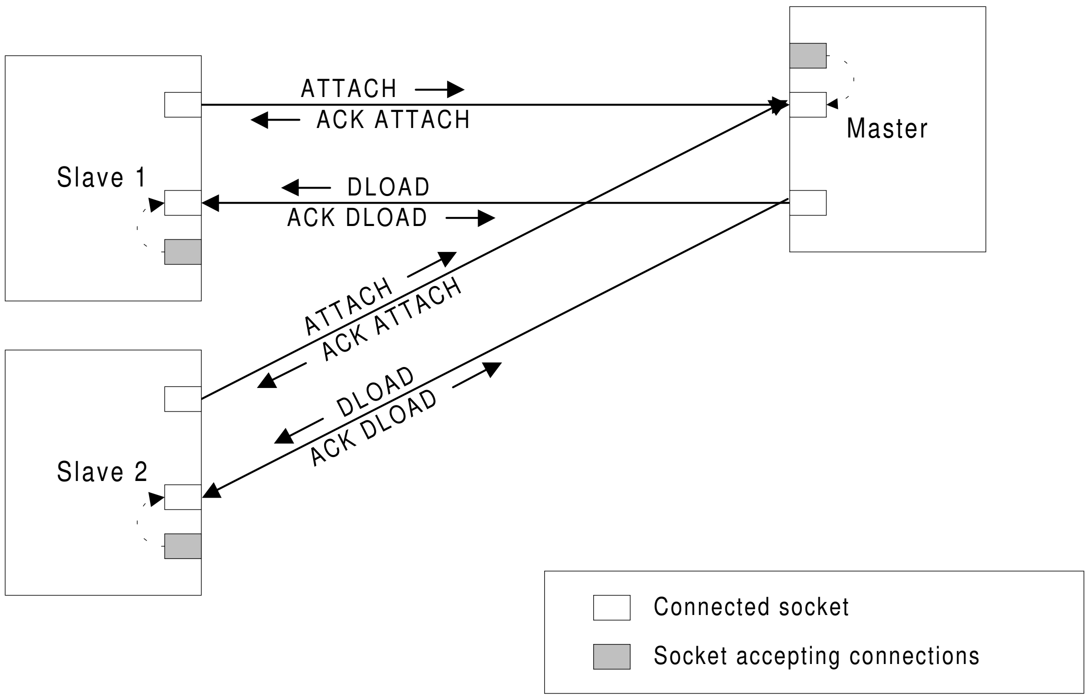
The top layer is essentially the original TOP-C system. It has been described in section 2.2.

3.1 General operation of lowest layer

The setup for the system consists of three processes:

1. the master process, linked with the TOP-C library and MPI. This program runs on the master computer, and manages the whole computation process;
2. the downloader process, which is started on a slave machine by a donor of CPU cycles, and then allows the master to download and start the application slave code; and
3. the application slave process, also linked against the TOP-C library. This program performs the actual computations on the slave machines. It is downloaded to the same machine where the downloader process is running.

A diagram of the protocol between two donwloaders and a master before the slave processes start running is provided below.



In the current design the master and slave programs are written in SPMD style and share the same source code and binary, and a command line switch is used to distinguish between the master and slave modes of operation. In general, the slave machines could be a different type of computers, and the slave binary will be automatically selected for download for each slave's architecture. However, the heterogeneous version of this software has not yet been implemented. The master process runs on the master processor throughout the computation. It automatically makes use of slave computers when they become available, and up to when they go off-line.

To be able to lend a machine to an ongoing TOP-WEB computation, the user has to install the downloader on his own machine. Then the user starts the downloader. The downloader connects to the master process. The master downloads and starts the appropriate binary on the slave machine.

It is not trivial for the master to generate the "appropriate" binary. In a homogeneous network, the master calls `unexec()` (see section 4) to generate a binary file containing the current environment. The master then sends the appropriate binary file to the slave downloader and the slave downloader then starts the binary. In a future, heterogeneous version, the master may have to first send a request to a slave of the right architecture in order to generate an appropriate binary via `unexec()`.

In the current implementation, all further communication between the master and the slave application program is done through the MPI layer, and does not use the downloader. The master adds the slave information to it's tables, and the slave becomes available for computations.

Thus, to reclaim the slave machine from usage by TOP-C, the user would simply kill the downloader and the TOP-C application. A computation task would likely be incomplete at that time, and the master would then send that task to another slave as one becomes available.

3.2 Downloader control protocol

The downloader control protocol uses ASCII strings for messages. The format is such that the messages can be manually created with a standard TELNET client for debugging purposes. The format for the messages is:

```
<command> { <flag>
          |<parameter key> = <parameter value> }
```

The words within the message are delimited by whitespace characters. The string is terminated by a newline character, which may be followed by additional binary data. The carriage return character is interpreted as whitespace, which allows for proper inter-operation with the TELNET protocol.

Each command has its own set of possible flags and parameters. The value of a parameter is an arbitrary text string, which is further interpreted in certain cases — for example the case of an integer in ASCII. The string may be enclosed in double quotes, in which case whitespace characters may be used in the string.

Throughout its run the master process accepts connections on a "well-known" TCP port. When the downloader is started on a slave machine, it notifies the master of another randomly chosen TCP port on which it will accept connections. Thus both the master and the downloader accept connections on some port as long as they are running.

A "conversation" between the master and the downloader may be initiated by either party. Each conversation consists of a message sent by the initiator, and a reply message, sent back by the receiver.

When one of the parties initiates a conversation, it opens a TCP port, which it connects to the other party's port. Then it sends the message, and receives the reply on the same socket. When the conversation is complete, the socket is kept open for some time, and subsequent conversations, initiated by the same party, reuse it. Note that while the downloader knows on which port the master is accepting the connections from the downloaders, the master doesn't initially know on which ports the downloaders accept connections. The downloader port is allocated dynamically, each time a downloader starts, and the downloader tells the master the port number, using the `PORT` parameter of the `ATTACH` message.

Note that each downloader process may have open at most two sockets: one for ATTACH messages initiated by the downloader, and one for DLOAD and other messages initiated by the master process. This way each party knows whether to expect a new command message or a reply to the party's previous command, according to what socket carries the message.

Each of the sockets may be closed by either party at any time, in which case a new socket will be open when the need arises. The downloader closes sockets on an idle timeout. The master may close the sockets both on the idle timeout and when it runs out of available descriptors. Since no sockets need to be kept permanently open, the number of descriptors allocated to downloaders by the master need only be a small fraction of the number of simultaneous downloaders.

3.3 Downloader control protocol - specific messages

When the downloader starts up it first attempts to send the ATTACH message to the master. Since the downloader can be started only by a person donating CPU cycles, the downloader is designed to operate during multiple sessions in which new master processes are started. This is accomplished by the downloader periodically repeating the ATTACH message and allowing the master to decide if this slave has previously been recognized.

```
ATTACH HOST=<slave host name> PORT=<port number>
    ARCH=<machine architecture>
```

The master replies with:

```
ACK ATTACH
or
NACK ATTACH REASON=<error message>
```

The downloader re-sends the ATTACH message after some reasonable period of inactivity, such as 15 minutes or an hour, so that if the TOP-C master program were not running when the downloader first started, it would still let the master know of the slave's availability once the master program starts. Of course, each instance of ATTACH message would typically require opening a new socket, because the socket "recycling" timeout would normally be shorter than the ATTACH repeat interval. Each time the master receives an ATTACH message, it has to check whether the message is from an already known slave, and if not, the master records the new slave's information. In a future heterogeneous version, there will be an ARCH parameter to allow using slave computers with different architectures. The master will automatically select the appropriate application binaries for download.

The downloader understands a few simple commands, such as:

```
DLOAD [ NAME=<application name> ]
    SIZE=<file size>|}
<binary data>

START [ NAME=<application name> ]

STOP [ NAME=<application name> ]
<binary data>
```

The DLOAD command serves for downloading the application binaries. The SIZE parameter let's the downloader know the size of the block of binary data, that follows the command string. The data is saved to a file on the local disk. The START command starts the application binary, and the STOP command stops (kills) it. The STOP command may be used to recover in case of a run-away application.

The NAME parameter in all the above commands is currently unused, though the downloader understands it. Theoretically, it would allow having several applications downloaded to the slave machine at the same time, even though only one of them could be running at any time. The downloader makes sure that the NAME parameter specifies a file in the current directory. The default is `top-c-app`, and the current implementation always uses it.

The slave would reply to each command with:

```
ACK <command name>
or
NACK <command name> [ REASON=<error message> ]
```

The error message returned for unsuccessful commands is a human readable string that can be used for debugging and analysis.

4 Long-Running Jobs: Checkpointing

A particular strength of TOP-WEB is its support for automatic checkpointing for long-running jobs. This is anticipated as an important requirement for such Web metacomputing since very large computations may require weeks or months of publicity before attracting sufficient voluntary Web donations of CPU cycles to make appreciable progress. A job that cannot checkpoint itself may find that it has died an unnatural death due to random, external causes before it has a chance to gather enough publicity to attract many donors of CPU cycles.

The ability to easily checkpoint a long-running computation comes from the task-oriented nature of the computation. The strategy is quite simple. When the master process wishes to checkpoint its state, it simply stops issuing new tasks from within `get_task()`. (To be more precise, it returns the special constant, `NOTASK`.)

Eventually, after at most n `REDO`'s and calls to `update_environment()`, where n is the number of slaves, there will be no more outstanding activities. At that point, the master process contains a copy of the same environment as each of the slave processes. So, it suffices for the master process to call `unexec()` to dump its current state to disk as an executable file. The GNU file, unexec.c, currently distributed as part of GNU emacs, does exactly this.

That executable file can then be called again, and it will begin executing at the point in the code where `unexec()` had been called. It will begin executing with the same state (same data and stack segments) as had existed when `unexec()` was called. Further, this executable file becomes the new downloader file that will be sent to any new slave processes. In the meantime, old. extant slave processes continue to wait for a new task as a part of their work loop. Hence, no special actions need to be taken with respect to such old slaves.

This checkpointing scheme has certain implications for heterogeneous workstations and PC's. Clearly, if the slave is of a different architecture than the master, then something special has to be done if these different slave architectures are to continue in the computation.

The heterogeneous capability of TOP-WEB has not yet been implemented. Implementation of this feature for heterogeneous workstations/PC's is planned, but it will require one additional restriction. TOP-WEB must have access to a slave of each architecture being used, at the time that it checkpoints. In this case, one slave process from each unique architecture is sent a special message at the lowest protocol layer, asking it to `unexec()` and send a copy of that executable file to the master process. That slave must then delete the local executable file that it created as a result of `unexec()`.

5 Application performance

The implementation of TOP-WEB performs with no noticeable difference from the static master-slave topology of the TOP-C system. Section TOPC-results reports on several long-running computations under TOP-C on a local area network. Aside from the formulation of the application, itself, the major contributing factors to performance in TOP-WEB are the overhead for each task and the startup time to attach a new slave.

These times were measured on a local area network using standard Ethernet with a theoretical maximum bandwidth of 10 MB/s and a lower practical bandwidth. The master was on one subnet and five slaves were attached on five remote processors, all on a second subnet connected by a switch. Experiments were run late at night, when the network and workstations were mostly idle. The six workstations involved were all 170 MHZ UltraSparc-1's. The GNU gcc compiler (version 2.7.2.3) was used with no special compiler flags set.

The size of the required libraries is 345 KB, split as 45 KB for the TOP-C model, 208 KB for the MPI subset implementation, and 92 KB for the portion of the loader protocol integrated with the master. In addition, the binary for the loader process requires 78 KB. The number of lines of C code is 1400 for TOP-C, 1450 for mpinu, and 1800 for the loader protocol in the master. The binary for the loader process requires an 300 lines of code in addition to the protocoal code that is shared with the master program.

To measure the times, we chose a naive application for factoring integers. The number, 50,000,017 (a prime) was attempted, and all smaller integers were tested up to the number to be factored, itself. The program does not stop testing when it reaches the square root of 50,000,017. Typical times for each run were about 20 seconds. A task consisted of testing TASK_INTERVAL consecutive integers. The methodology was to vary TASK_INTERVAL between 5×10^4 and 2×10^6 (and hence the number of tasks) and observing how the overall real running times varied in attempting to factor 50,000,017. At larger values, the number of tasks was not much more than the number of slaves, resulting in uneven load balancing. At smaller values, network congestion due to many simultaneous tasks resulted in unrepresentatively long times. Times found were reproducible.

Within this range, the times (in seconds) for the different values were as follows. From this data, we conluded that there was an overhead of 70 milliseconds per task, based on communication overhead and slaves remaining idle until the next task could be produced. We suspect that the time is mostly accounted for by the former in this application, since the master used only about 2% of the master processor CPU time.

TASK_INTERVAL	5×10^4	1×10^5	2×10^5	5×10^5	1×10^6	2×10^6
Time (s)	27	17	12.4	12.4	12.4	12.4

In order to find the startup time, we started a master job, and then started 16 jobs on 16 slave processors. As before, the slave processors were on a second subnet. Because the computation was so fast with 16 slaves, the number to factor was raised to the prime, 1234567891, which is within a factor of two of the largest positive 32-bit number. No attempt was made to optimize the code, since the startup times were the primary interest. The master acknowledges the requests for new slaves in sequence rather than in parallel. The time for starting the new slaves is $16 \times 4 = 64$ seconds. The remaining phase is the actual computation and requires an additional 50 seconds.

Finally, one can give anecdotal evidence related to a long-running TOP-C example. For such a long-running job, we would not expect the TOP-WEB performance to differ from that of TOP-C. The application that had stressed the original TOP-C (or LISP/MPI, in this case) implementation the most, to date, was the condensation of Th [9]. This application ran for one month on eight DEC Alpha 300's. Under these circumstances, a typical task would run for twenty minutes after which the slave process would send a result message of about 20 Megabytes. The result message required about one minute of real time. This seemed to occur both under standard Ethernet and when using a half-gigabit per second DEC switch. From this, we concluded that the bottleneck was probably the interface to the workstation, itself.

6 Comparison of TOP-WEB and Charlotte

In comparing TOP-WEB and Charlotte, the most immediately striking difference is that TOP-WEB is currently written in C while Charlotte is written in Java. In fact, we had originally wished to develop TOP-WEB in Java, but were constrained by legacy code. Although it is possible for Java applications to call native, binary code through a standardized API, this violates the Java "sandbox" security model for applets. Specifically, this is not an option for applets distributed over the network. Further, the technology does not currently exist for translation of C to Java. Similarly, the technology does not currently exist for compilation of C code into Java byte code for the Java virtual machine. Finally, Java was rejected for the present because the current state of Java compiler technology results in a large performance gap between Java and C. For all of these reasons, we decided upon an initial C implementation and a later port to Java.

Further, the lack of the Java "sandbox" implies that the system is not running securely. So, currently someone will usually donate CPU cycles only if he/she knows about the application and considers it to be a trusted applications for purposes of downloading. Something similar to this takes place very frequently on the Internet when people download free games. Nevertheless, the issue of security would soon be a limiting factor, and we anticipate porting to Java when Java compilers become faster.

Indeed, the presence of Charlotte now opens up to us an alternative in which TOP-WEB would be implemented as a task-oriented parallel model running in Java on top of Charlotte in order to obtain the advantages of both models. This is possible because Charlotte provides a very general, distributed shared memory model. There might also be opportunities to incorporate Charlotte's more sophisticated scheduler into TOP-WEB.

In our opinion, the deeper distinction between Charlotte and TOP-WEB is the choice between Charlotte's language-level distributed shared memory and TOP-WEB's task-oriented parallelism. In general, the distinction can be viewed as one of deciding at what level to parallelize an application. For example, data-oriented parallelism tends to result in parallelization at a much lower level than task-oriented parallelism. A distributed shared memory system, such as Charlotte, tends to be neutral about what level should be parallelized. In contrast, TOP-WEB actively encourages the application programmer to re-organize his/her program so as to be able to parallelize at the level of higher level tasks. This active encouragement towards task decomposition results in a somewhat greater burden for the application programmer, but it also results in greater efficiency and more opportunities for overlapping communication and computation, resulting in good latency hiding.

The two systems can also be contrasted with respect to their strategy toward moving data across the network. In this respect, TOP-WEB may have an advantage. As the developers of Charlotte point out, "Our general implementation [of distributed shared memory] is not particularly efficient, but simple and tractable." [1, Section 5.2] In contrast, the TOP-C model tends to call for less frequent network communication, since communication occurs only at the beginning and end of each task. This bundling of many messages by TOP-C has certain efficiency advantages for communication. Nevertheless, it should be pointed out that Charlotte's general model does offer the possibility of emulating the TOP-WEB architecture. Under these circumstances, Charlotte could take advantage of bundling of data movement together at the time of task completion. This is, in fact, the merging of the technology of Charlotte and TOP-WEB referred to above.

In general, there may be no single answer as to which system has better performance. For example, Charlotte would tend to have an advantage where there is little data movement, as with the application of Charlotte to the 3D Ising Model described in [1]. In that case, the TOP-WEB applications programmer would have to go to some additional effort to bundle several tasks together so as to avoid excessive overhead in communication between master and slave during task initiation and completion. On the other hand, TOP-WEB would tend to have an advantage in situations where several slave processes are accessing and updating a single shared data structure, since the TOP-C model was designed precisely with this case in mind.

In general, each system would tend to suffer in certain kinds of scaling. For example, TOP-WEB would have difficulting scaling the number of processors, since the master process then becomes overloaded. Charlotte would have difficulty scaling as the amount of communication or data movement rises. In this latter case, TOP-WEB has two advantages. Not only does it postpone data movement until completion of a task, thus allowing bundling into a smaller number of packets. But TOP-WEB also has the possibility of using application-dependent strategies so that a minimal amount of data is passed across the network, perhaps in a compressed format, and then fed to each slave process as input to `update_environment()`. Such a strategy may yield a beneficial time-communication tradeoff.

7 Why not in Java yet?

The ideal system of the form described above would be written in Java. Java or some similar technology will be required to assure security to the donor of CPU cycles. The security issue is paramount if this and similar systems are to become widespread. However, Java is not yet well enough accepted for this to be a currently practical option. Hence, the current implementation has been done in C. Nevertheless, all of our design decisions were made so as to be compatible with a future port of this system to Java. So, one may view this implementation both as a currently functional implementation and as proof of concept for a future Java version.

The Java model is ideal due to its "sandbox" that isolates a Java applet from user resources. However, this would require that the application be written entirely in the Java language. This imposes a burden for two reasons.

First, applications programmers typically write scientific applications in C, C++, or FORTRAN. It is true that Java defines an API for running native code on a processor. However, the Java security model forbids downloading of native code, since such native code would not be bound by the security model enforced by the Java interpreter, and it could access arbitrary user resources.

Second, Java code currently runs too slowly. This situation is likely to be improved as the technology of Just-In-Time compilers (JIT compilers) improves. Further, SUN has announced that Java CPU chips will be available both for network computers and as add-in boards for PC's. Thus, many of the free CPU cycles of the future are likely to be available from such Java chips.

Particularly in scientific applications, an issue for Java is that its security model requires array bounds checking. Since arrays feature prominently in most scientific applications, this would be a severe restriction. Luckily, current CPU technology often encompasses instruction level parallelism (ILP) [14]. Thus array bounds checks can often be executed in parallel, asynchronously to the main instruction stream. One would presume that efficient implementations of Java CPU chips will also implement this feature.

There is one more issue in eventually using Java. This is that one would have to implement an analogue of `unexec()` for Java, or else keep track of previous calls to `update_environment()` and re-invoke them on a new slave or a restored checkpoint. This issue has not currently been addressed.

8 Conclusions and Future Work

The primary issues that we have to be concerned with respect to the TOP-WEB development are security and performance. While the current implementation of TOP-WEB addresses the performance question quite well, some security issues have not been resolved. The TOP-WEB model is based on the ability to download binary code to the participants' machines and run it there. Furthermore, the current implementation of the download mechanism requires that the downloading and running of the TOP-WEB application binaries were done automatically. While this would be acceptable on a well set up workstation, with the TOP-WEB code running out of a "guest" account, it would create an additional security hole on a participant's system that has not been set up so well. For example in a time-sharing system, the original master might have died, and malicious software could begin listening on the same listener port, posing as the master. A facility is being added for the loader to authenticate the TOP-WEB master, to avoid the risk of such software posing as the master and downloading its own software. Since the loader and the master binaries are initially created on the same machine, a good solution is for the loader binary to ask a challenge question that only the true master software can answer. A future Java implementation will reduce the security issue to that of the Java sandbox security model, which will hold greater confidence.

The current version of TOP-WEB is distributed as part of the TOP-C distribution at `ftp://ftp.ccs.neu.edu/pub/people/gene/top-c/`. The total size of the libraries created on the UltraSparc-1 using GNU gcc is 345 KB. The size of the downloader binary is 78 KB.

References

[1] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, Charlotte: Metacomputing on the Web, *Proc. 9th Intl. Conf. on Parallel and Distributed Computing Systems*, 1996.

[2] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms, *Proc. 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.

[3] G. Cooperman, "Practical Task-Oriented Parallelism for Gaussian Elimination in Distributed Memory", *Linear Algebra and its Applications*, to appear.

[4] 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.

[5] 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.

[6] 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/`).

[7] G. Cooperman, L.Finkelstein, M.Tselman and B.York, Constructing Permutation Representations for Matrix Groups, *J. Symbolic Computation*, to appear.

[8] 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.

[9] 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*, to appear.

[10] 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.

[11] R. Daoud and G. Burns, New Features and Old Principles in LAM 6.0, MPI Developers Conference, June 22–23, 1995, University of Notre Dame, `http://www.cse.nd.edu/mpidc95/`.

[12] 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*.

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

[14] Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, second edition, 1996.

[15] R. Lusk, N. Doss, A. Skjellum, and W. Gropp, MPICH: A High-Performance Portable Implementation of the MPI Standard, MPI Developers Conference, June 22–23, 1995, University of Notre Dame, `http://www.cse.nd.edu/mpidc95/`.

[16] MPI Forum, MPI 2 Home Page, `http://www.mcs.anl.gov/Projects/mpi/mpi2 /mpi2.html`.

[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.

[18] L. Vanhelsuwé, Create your own supercomputer with Java, *Java World*, January, 1997 (see also `http://www.javaworld.com/javaworld/jw-01-1997 /jw-01-dampp.html`).