DSC
A System for Distributed Symbolic Computation*

A. Diaz, E. Kaltofen, K. Schmitz, and T. Valente
*with contributions from: M. Hitz, A. Lobo, and P. Smyth***

Department of Computer Science, Rensselaer Polytechnic Institute
Troy, New York 12180-3590; Inter-Net: kaltofen@cs.rpi.edu

DSC is a general purpose tool that allows the distribution of a computation over a network of Unix workstations. Its control mechanisms automatically start up daemon processes on the participating workstations in order to communicate data by the standard IP/TCP/UDP protocols. The user's program distributes either remote procedure calls or source code of programs and their corresponding input data files by calling a DSC library function. DSC then automatically finds a suitable workstation and sends the information necessary to execute the given subtask. If source code, which may be written in either ANSI C or Common Lisp, is distributed, this includes compilation on the remote workstation to first make an executable object module. In this mode, a Unix workstation of any architecture type or without prior preparation of problem specific object modules on its file system can be involved. A call to another library function synchronizes the user's program execution by selectively waiting for the completion of a specific or all distributed subtasks. DSC also has an interactive monitor facility that lets a user watch the progress of a distributed computation. We have tested DSC with a primality test for large integers and with a factorization algorithm for polynomials over large finite fields and observed significant speed-ups over executing the best-known methods on a single workstation computation. These experiments have been carried out not only on our local area network but also on off-site workstations at the University of Delaware.

1. Introduction

With the advent of large numbers of workstations linked together by a programmable network it becomes possible to distribute a large scale computation over many workstations. DSC is a software system for the distributed processing of a large computer algebra computation. The DSC system currently runs on workstations with the Unix operating system communicating via the DARPA Internet standard IP/TCP/UDP protocols. Ours is a heterogeneous approach to problem solving in the following sense: DSC supports the distribution of C as well as Lisp code, it has been tested on several architectures, namely Sun 4, Sparc, and DecStation computers, and it distributes over our local area network as well as over the Internet to off-site computers at the University at Delaware.

The system hides processor scheduling and the actual remote execution of a parallel subtask from the user. In fact, when a user program supplies the appropriate library function with a distributable parallel subtask, the local DSC server daemon process will search a database for a suitable processor, depending on work load and resource availability, and then distribute the parallel subtask to the selected processor. It is not assumed that the program code necessary to execute the parallel subtask resides on the file system of the selected remote processor. The DSC system can send the C or Lisp source code and input data files that constitute the parallel

*This material is based on work supported in part by the National Science Foundation under Grant No. CCR-87-05363 and under Grant No. CDA-88-05910, and by the National Security Agency under Grant No. MDA904-90-H-4003.

**Supported by the National Science Foundation as an undergraduate summer research student under Grant No. IRI-87-12838.

In Proc. Internat. Symp. Symbolic Algebraic Comput., S. M. Watt (ed.), pp. 323–332 ACM Press (1991).

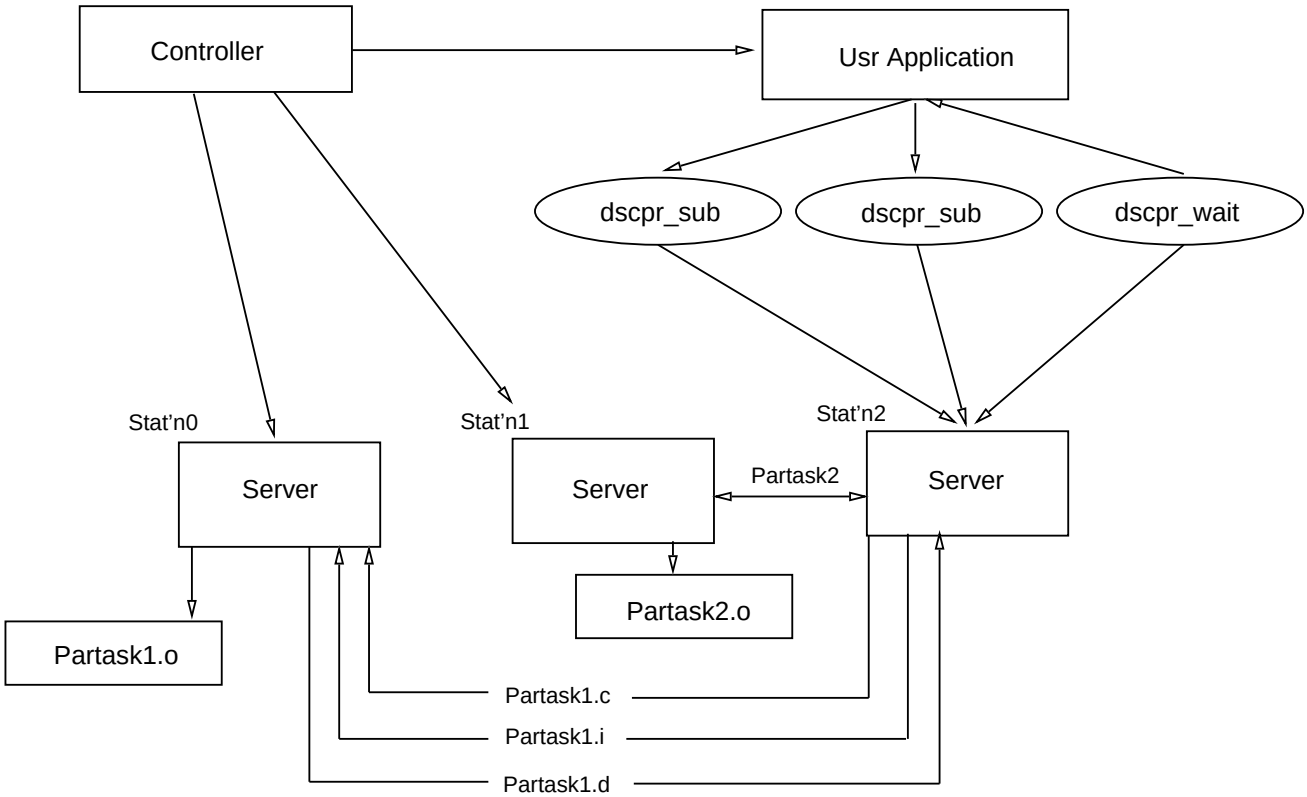

Figure 1: Sample configuration; rectangles denote processes.

subtask to the remote processor, whose DSC server daemon process in turn will compile and run the received program on the input file. The parallel subtask is supposed to produce a corresponding output file, which will be returned to the sender for the collection of results. In an alternate mode of operation the executable program already resides on each selected workstation, and only the input file is sent. That mode essentially realizes the remote procedure call distribution mechanism. Note that the parallel subtasks themselves can spawn further parallel subtasks, which in turn are distributed by the remote servers.

Before a user starts an application program that distributes parallel subtasks over the network, the database of available workstations needs to be initialized and the server daemons started. This can be done interactively in an automated fashion using the DSC Controller program. The controller also allows the monitoring of a computation on all used workstations. In particular, status logs of running or failed parallel subtasks can be interactively inspected. Processors that cease to function during the life of a computation are abandoned without jeopardizing the overall completion. It is even possible to kill a hanging parallel subtask and restart it on a different processor.

Figure 1 above depicts a simple scenario of a distributed run. A user has started the controller on workstation 0 and then aided by a controller menu selection has started servers on workstations 1, 2, and 3. This user then executes an application program, which distributes a parallel subtask. The corresponding DSC library function `dscpr_sub` is called with both a C language source file, `partask1.c`, and the file containing the input data, `partask1.i`, both of which have been created by the user program. The function `dscpr_sub` communicates to the local server running on workstation 0 (also started by the controller), which then locates a suitable processor, in this case workstation 1, and sends both files to the corresponding remote server. The server on workstation 1 compiles the remote copy of file `partask1.c` and executes it. That program then reads its data from the remote copy of `partask1.i` and writes its output to the remote file `partask1.d`. The file names are deduced from arguments on the command line issued by the server on workstation 1. In the meantime, the application program has initiated distribution of another parallel subtask, partask2, for which the local server selected workstation 2. It then has issued a `dsc_next` library function call, which causes a wait until one of the two parallel subtasks has completed; again it is the local server that will process the messages coming from the remote servers, diagnose this

situation, and return control and the necessary information on which parallel subtask has completed to the application program. A remote server recognizes completion of its parallel subtask by repeatedly probing the status of the executing process. If a parallel subtask completes without error, the file `partask1.d` is copied back to workstation 0. The files on all remote workstations are eventually removed by the servers.

The design and implementation of the DSC system was partially motivated by the success of distributed computation on the problem of factoring integers (Lenstra and Manasse 1990), partially by the obvious coarse grain parallelism in symbolic methods based on homomorphic imaging such as Chinese remaindering (Seitz 1990), and partially by our new algorithm for factoring polynomials in black box representation (Kaltofen and Trager 1990). The latter method produces for each multivariate irreducible factor a program that when supplied with values for the variables computes the value of the polynomial factor. If one wishes to apply a sparse interpolation algorithm (Kaltofen et al. 1990) to such a black box factor, one can obtain the values of the polynomial at the necessary interpolation points by distribution. Of course, one can also distribute the interpolation of each factor. This was our primary reason for requiring that the DSC system can distribute source code, which in the case of the black box polynomial factorization algorithm is computer generated. It turns out that this requirement is very useful in general, since it alleviates the user from setting up the parallel subtask code on every possibly used workstation. Thus, in a distributed computation, workstations of different hardware architectures can be freely mixed.

At the moment, however, we exercise DSC on the distributed implementation of two other algorithms. The first is our version (Kaltofen et al. 1989) of the Goldwasser-Kilian/Atkin primality test (Atkin and Morain 1990), with the reduction phase done in C and the certification phase done in Lisp. The second is a distributed Lisp implementation of the Cantor-Zassenhaus polynomial factorization algorithm over finite fields of large characteristic. We describe the details of these solutions in §4, but we like to point out already that the distributed approach led to a remarkable change of approach in the Cantor-Zassenhaus algorithm.

Discussion of relation to other efforts: The use of parallelism in computer algebra was studied in the Ph.D. theses of Watt (1986), Ponder (1988), Roch (1989) (see also Roch et al. (1988)), and Seitz (1990). Implementation of computer algebra systems in parallel computing environments were done by Melenk and Neun (1986) for Reduce on Cray model computers, by Johnson (1988) for SAC-2 also on Cray's, by Saunders et al. (1989) for SAC-2 on Sequent computers using the Linda environment, by Collins et al. (1990) also for SAC-2 on the Encore Multimax, and by Char (1990) for Maple, using the Linda environment. Seitz (loc. cit.) has implemented a distributed version of SAC-2 on Sun workstations. We know of several efforts to provide general purpose distributed computing environments, among them the systems by Silverman and Stuart (1989), by Sullivan and Anderson (1989), and the ISIS project at Cornell University. One innovative feature of DSC is the ability to distribute source code, thus making the programmer oblivious to the configuration of the processor network that is used. Also, a DSC parallel subtask is a first class object in that it can distribute its computation further.

In §2 we describe the DSC system from a user's point of view, while in §3 we discuss some of the intricacies of the underlying system layers. In §4 we discuss our two current applications and in §5 we point to future development plans.

2. User Interface

The utilities described in this section serve as fundamental tools, which allow a user to program the activation of parallel subtasks, wait for their completion, and optionally manage their execution.

2.1. The C User Library

The C User Library contains 5 functions callable from a user's C program. The function `dscpr_sub` is used to start a parallel subtask within the DSC environment. Arguments to this function are 4 strings denoting, respectively, the path to the local files, the types of the local files to be sent, and the build, e.g., compile, and exec, e.g., load and run, commands required for the parallel subtasks. It is possible to use default build/exec commands by supplying null strings for the corresponding arguments. It is also possible to bypass compilation by supplying `dsc_c.build` as the build command.

The functions `dscpr_wait` and `dscpr_kill` are used, respectively, to wait on and to kill specific subtasks. In both cases, the index of the subtask in question is supplied as argument. The latter function is also used to wait on the completion of *all* outstanding subtasks when a value of -1 is supplied for the index. The function `dscpr_next` is used to wait for the completion of the next parallel subtask. Its single argument, a string, is filled with the name of the solution file corresponding to the completed subtask. Finally, the function `dscdbg_start` can be used to track a problem and is useful when one wishes to debug tasks using interactive debuggers such as Unix's dbx.

The use of these functions is illustrated in Figure 3 below.

2.2. The Common Lisp Interface Functions

The Common Lisp Interface contains 5 functions which

are analogues of the C User Library functions, as well as 2 additional functions. The function `dscpr_sleep` is used to suspend execution for a specified number of seconds. The function `dscpr_time` can be used by the user to compute the elapsed time of a parallel subtask. A fragment of Lisp code using these functions is depicted in Figure 4.

2.3. Controller Program

The DSC controller program is an interactive program which can be used to start new applications, monitor the progress of applications and parallel subtasks, terminate applications and halt DSC activity on a given node. Once started, the controller program will display the menu shown below.

```
******* DSC Control Menu *******
 1:     Start new problem
 2:     Display Problem Database
 3:     Display Node Database
 4:     Display Server Statistics
 5:     HALT Server on this node
 6:     Kill a specific subproblem
 7:     Start single remote Server
 8:     HALT all configured Servers
 9:     Start all configured Servers
99:    Exit this menu
option:
```

We now describe the possible selections in the order one would use them in running a program that distributes parallel subtasks. First we discuss those menu items associated with managing servers.

Before an application program can commence, a DSC server must be started on all of the nodes participating in the application. Option 9 will start all configured DSC servers. Option 7 can be used to start a DSC server process on a single remote node. The node in question must have the DSC package installed and be configured into the `DSC_PATH/dscrp_net.lis` file. This file lists, on separate lines, information for each node participating in the DSC network. Each line consists of a node name, followed by two integers indicating the Unix nice value for the DSC server and parallel subtasks, respectively. The remaining three fields indicate which languages are supported by the node in the order C, Lisp, and Mathematica. The conventions are: `c` for C language support, `l` for Lisp language support, `m` for Mathematica, and `x` if the corresponding language is unavailable. An example of this file is as follows:

```
apollo.cis.udel.edu|0|1|c|l|x
gemma.cs.rpi.edu|0|1|c|l|x
turing.cs.rpi.edu|0|1|c|l|x
vega.cs.rpi.edu|0|0|c|l|m
mays.cs.rpi.edu|0|1|c|x|m
```

Note that no information on the hardware architecture of each node is given. When compiling the server and DSC system library code on each individual workstation, this information is determined and set. Option 5 will halt the DSC server on the local node. All applications and parallel subtasks executing on the local node will be terminated. All remote DSC servers that have parallel subtasks operating on this node will be informed of the forced termination. Option 8 can be used to terminate all configured DSC servers. This function will send a message to each server requesting that it shut down in an orderly fashion, thereby updating the logs properly.

Option 1 of the main menu allows the user to execute an application program. The user must interactively supply a name for use in the DSC database, the source file name, and the type(s) of the source file. Optionally, for C source, the user can specify a build instruction and an execute instruction, while for Lisp source, only an execute instruction can be given. If not supplied, a default instruction is used. Figure 2 shows how to use this option.

The control program also allows the user to inspect the progress of an application program and its parallel subtasks. Option 2 of the main menu will generate a display of the parallel subtask database. This will include the main application program and the parallel subtasks that have been spawned in the current DSC environment. Figure 2 is an example of the parallel subtask database. Notice that the main Lisp program, prefixed as `canzass` and running on `turing`, has distributed Lisp object code to `vega` and `gemma`. This is indicated by both `lisp: 1` and `obj: 1` being set. The shell script `dsc_lisp.obj` was used in the exec argument of the call to `dscpr_sub`. Neither parallel subtask has spawned children. All processes run on Sun workstations, as `CPU 3` indicates.

DSC also allows for the monitoring of servers on the local node as well as remote nodes. Option 3 will display the node database. This includes the status of each remote configured node and its current load. Option 4 will display the load statistics for the local node. Option 6 can be used to terminate a single parallel subtask. The user is asked to supply the index of the parallel subtask to be terminated. This index is found in the parallel subtask database.

3. System Layers

In the previous section we described how one can employ the user interface for the distribution and monitoring of parallel subtasks. However, this is only one segment of what can roughly be viewed as a hierarchical organization. Conceptually, the system itself is organized in a multi-layered fashion, each layer drawing from its predecessor. The bottom layer consists of the interprocess

```
        option: 1
Enter New Problem name: Cantor-Zassenhaus
Enter Source File Name (no extention): /students/ugrads/diaza/dsc/canzass
Enter file type (c/l/lc/lo/i/ci/li/lci/loi) [c]: li
do you have special execute instructions ? no

     file: /students/ugrads/diaza/dsc/canzass
build cmd: dsc_lisp.build
 exec cmd: dsc_lisp.int /students/ugrads/diaza/dsc/canzass.lsp /students
/ugrads/diaza/dsc/canzass.d /students/ugrads/diaza/dsc/canzass.i

        option: 2
 problem name: Cantor-Zassenhaus
 problem  PID:  22313   node: turing.cs.rpi.edu index   0       CPU  3
 parent   PID:  22272   naddr: 128.213.1.1      index  -1
 source  name: /students/ugrads/diaza/dsc/canzass
           c: 0  lisp: 1  obj: 0  input:1
build command: dsc_lisp.build
 exec command: dsc_lisp.int /students/ugrads/diaza/dsc/canzass.lsp /students
/ugrads/diaza/dsc/parcant_obj.d /students/ugrads/diaza/dsc/canzass.i
 blocked sock:  -1       block condition -1
 completion  :   0      status: 00000000 build:   1
 peer link   :  -1      child list:   -1         # subprob:   2

 problem name: DSC572a1
 problem  PID:     -1   node: vega.cs.rpi.edu   index   1       CPU  3
 parent   PID:  22313   naddr: 128.213.1.1      index   0
 source  name: can1
           c: 0  lisp: 1  obj: 1  input:1
build command: dsc_lisp.build
 exec command: dsc_lisp.obj can1.o can1.d can1.i
 blocked sock:  -1       block condition -1
 completion  :   0      status: 00000000 build:   0
 peer link   :  -1      child list:   -1         # subprob:   0

 problem name: DSC572a2
 problem  PID:     -1   node: gemma.cs.rpi.edu  index   2       CPU  3
 parent   PID:  22313   naddr: 128.213.1.1      index   0
 source  name: can2
           c: 0  lisp: 1  obj: 1  input:1
build command: dsc_lisp.build
 exec command: dsc_lisp.obj can2.o can2.d can2.i
 blocked sock:  -1       block condition -1
 completion  :   0      status: 00000000 build:   0
 peer link   :  -1      child list:   -1         # subprob:   0
```

Figure 2: Interactive queries to controller.

communication using DARPA Internet standard protocols IP/TCP/UDP. Built on this layer lies the first DSC level which includes the internal DSC routines, the daemons, and the C library functions. The second layer consists of the Lisp/C interface, the controller program, and C user programs that use only the four basic C library functions. The third layer draws on the Lisp/C interface. This layer contains the Lisp library and basic Lisp functions that use only the six basic functions supplied by the interface. At the topmost layer lies the implementation of high level Lisp functions which utilize the support routines contained in the Lisp library. Examples of such routines are string and integer manipulation functions and functions for managing the ab-

stract data type Bag of Incomplete Futures (*BIF*) (cf. (Seitz 1990)) which will be discussed later in this section.

3.1. Interprocess Communication

DSC uses the User Datagram Protocol for most of its communication and Transmission Control Protocol stream sockets for more sensitive information. The DSC server executes on each participating node in the network. The function of this daemon program is to monitor a single UDP datagram address for new external stimuli from other servers. As messages arrive into this datagram socket, they are read and processed. If the request comes from the control program, the server forks a subprocess to begin the execution of the main task. At some point the main task will ask the server to start parallel subtasks, using the `dscpr_sub` call. This call issues a start message using a datagram causing the DSC server to send messages to an available server allowing it to initiate the execution of the specified parallel subtask and establishes TCP stream sockets for all of the parallel subtask's file transfer. The node selection is based on which nodes have started servers, what languages they support, and on the number of parallel subtasks executing on that node. If all of the available nodes have reached the parallel subtask node capacity limit, the parallel subtask will be put on a queue for later distribution. At this point the server checks the viability of the activated processor. If at any time these processors are declared non-functional the problem will be restarted elsewhere. Once the parallel subtask is complete the server responsible for its execution will forward the result to the originating server and delete the log files, source files, and executable files found in the `DSC_PATH`. Finally, once the main task ends the executable files in `DSC_PATH` and the log file in the users working directory are also deleted.

3.2. The Lisp/C Interface and Library

We use the standard KCL Lisp to Unix C interface, i.e., the `Clines` function. This function makes its C code argument callable from the KCL environment. The C code contains modified copies of the DSC C user library functions (see §2.1). There is also a set of Lisp utility functions that encode the string arguments of the Lisp user library functions (see §2.2) to the integer arguments in the interface functions supplied to `Clines`. Furthermore, there are functions to ease string manipulation, file copying, and the functions implementing the *BIF* data type. A Bag of Incomplete Futures consists of a list of unordered parallel subtasks. These parallel subtask can then be spawned by using the `dsc_bif` function which allows for the specification of the percentage of successful parallel subtask completions needed for a successful main task completion.

4. Current Experiments

4.1. The GKA Primality Test: First Stage

As our first example, we describe a distributed implementation of the first stage of the Goldwasser-Kilian / Atkin (GKA) primality test. Our implementation, written in C, uses a high-precision integer arithmetic package (HIPREC) developed locally by Hitz (1988). Detailed descriptions of the GKA test appear in (Atkin and Morain 1990) and (Kaltofen et al. 1989).

The input to the first stage is a number p which we suspect to be prime, together with a list of fundamental field discriminants $d_i < 0$. This list must be searched in an attempt to find a descent from p to a smaller probable prime. Thus, via a series of such descents, the first stage of the test constructs a sequence of probable primes $p = p_0 > p_1 > p_2 > \cdots > p_n$ such that p_n prime $\Rightarrow p_{n-1}$ prime $\Rightarrow \cdots \Rightarrow p_0$ prime. For large p, a descent can be difficult and costly to find, as nearly all of the discriminant list may need to be searched. It is therefore advantageous to allow several distributed subtasks to contribute to the search.

The idea is to conduct the search for a descent in parallel by partitioning the discriminant list into N sublists, where N is the number of parallel subtasks, and providing each parallel subtask with its own sublist as input. DSC is then used to run the parallel subtasks (each of which has the same code) simultaneously on different workstations, and to wait on the first subtask that reports a descent. This "first descent" strategy is especially effective when p is large and we cannot expect to find more than one or two descents. If, on the other hand, p has, say, fewer than 100 digits, the subtasks may collectively report several descents, allowing us the luxury of invoking a "steepest descent" strategy. Listed in Figure 3 are code segments from a simplified version of our program illustrating this strategy. Our current version implements both strategies and allows for backtracking. Also, we now avoid sending C source files to remote nodes and the subsequent builds.

4.2. The Cantor-Zassenhaus Factorization Algorithm

Our next example shows how DSC can give insight to a new approach to the Cantor-Zassenhaus (1981) algorithm for factoring a polynomial over a finite field. We suppose $f(x) \in \mathrm{GF}(p)[x]$ is a squarefree polynomial which factors as $f(x) = q_1(x)q_2(x)\cdots q_r(x)$ where the q_i are distinct irreducible elements of degree d of $\mathrm{GF}(p)[x]$. Given a random "trial" polynomial $t(x)$, it is likely that $\gcd(f(x), t(x)^{(p^d-1)/2} - 1)$ will be non-trivial, thereby causing a "split" of $f(x)$.

For our current DSC experiment, we further restrict ourselves to the case where the q_i are all linear, and for $1 \le i \le n$, we send the i-th parallel subtask $f(x), p$, and

```
/* include functions from HIPREC package ... (omitted) */
#define SOURCE_FILE "gkabest.c" /* contains code for subtasks */
main ( argc, argv) int argc; char **argv;
{int y,i,delta, some_descent_occurred=TRUE;
 /* various string, FILE, and HIPREC declarations ... (omitted) */
 /* get initial p to be proven prime from input file ... (omitted) */
 while (some_descent_occurred  && (gt(p,billion)) ) {
    zero(q); zero(biggest_k); zero(current_k);
    some_descent_occurred = FALSE;
    writefiles(NUMPROBS,p); /* create ".i" input files for subtasks */
    for (y=0; y<NUMPROBS; y++) {
       /* create subtask C source files; note that they will go
          into current working directory at run time */
       sprintf( fid, "gkasub%d", y); /* build source file name */
       strcpy( fidc, fid); strcpy( fidd, fid);
       strcat( fidc, ".c"); strcat( fidd, ".d");
       sprintf( build, "cc -o /var/tmp/gkasub%d %s", y, fidc);
       sprintf( exec, "/var/tmp/gkasub%d %s", y, fidd);
       /* copy SOURCE_FILE to file named in fidc ... (omitted) */
       /* start the subtask */
       dscpr_sub( fid, "ci", build, exec) } /* end for */
    /* wait on all subtasks to complete */
    i = dscpr_wait(-1);
    /* loop to read reported descents from all data files
       and determine q corresponding to steepest descent from p
       ... (omitted) */
    if (some_descent_occurred) {
       /* write k,q,delta to certificate file ... (omitted) */
       hp_copy(p,q); /* p = q */ } } /* end while */
 exit(0); }
```

Figure 3: Code for distributed primality test.

trial polynomial $t_i(x) = x + i$. Each subtask then computes the appropriate gcd and reports the split $f = g_i h_i$ accomplished by t_i, if any. Once enough parallel subtasks have reported, it is possible to use, one-by-one, the reported splits to ascertain the *complete* factorization of f by factor refinement (Bach et al. 1990). A fragment of the Lisp code for the distributed Cantor-Zassenhaus algorithm appears in Figure 4.

4.3. Timings

Figure 5 shows the elapsed times of repeated runs of the GKA first stage on three prime inputs of length 22 and 43 digits, executed with the indicated numbers of parallel subtasks. Several runs are timed for the same prime input, since the algorithm is non-deterministic in the sense that different first possible descent found may be reported back by those workstations that complete the search first. The case of 22 digits is too small to exhibit a benefit when using distribution. The first discriminant to cause a descent from our 43 digit prime does not occur until after 109 discriminants in our original list. Thus, with 1 workstation, we see running times in excess of 30 minutes for the seven descents, with more than 20 minutes required for just the first descent. Notice that a remarkable improvement occurs with just 4 workstations, as now this same discriminant occurs as only the 28th on some parallel subtask's list. However, corresponding improvements do not occur for 8 and 16 workstations, where distribution overhead seemingly dominates the cost of inspecting shorter lists. Figure 8 contains overhead costs *per descent* associated with the GKA test for varying numbers of subtasks. These timings were obtained by creating and distributing the appropriate number of GKA input files, executing null programs at each of the involved nodes, and waiting on the first program to complete.

An interesting phenomenon occurs in the data for our 201 digit runs: 16 workstations require more time to uncover a descent than do 8 workstations. This appears to be caused by the particular assignment of sublists to workstations. The data for 2 and 4 subtasks indicates that the DecStations discovered the descent about twice as fast as did the Sun Sparcs. In each case, the discriminant that yielded the descent was -763, which appears 80th on our original list. When this list is then par-

```
;; Interface header follows here ... (omitted)
    (load "dsc_lisp.interface") ;; Load DSC Lisp/C interface
    (load "dsc_lisp.library.o") ;; Load library
    (load "cantor.split.o")     ;; Load auxiliary applications functions
    ;; Driver program
    (defun cantorz (poly modulo)
      ;; "poly" is to be factored, "modulo" is the prime modulus
     (let ((r (list poly)) ;; initial value of refinement
           (name "can")    ;; prefix for file names
           (n 15)          ;; n is the number of subtasks
           ;;  ... additional "let" forms omitted
          )
      ;;; Create "can1.i","can2.i",...,"can<n>.i" file names using
      ;;; string functions from DSC Lisp library ... (omitted)
      ;;; For tout = "can1.i","can2.i", ... "can<n>.i"
      ;;; write poly, modulo and trial-poly to tout:
           (print tmp tout)
           (print poly tout)
           (print modulo tout)
      ;;; Loop to distribute follows below:
      (loop for i from 1 to n by 1 do
      ;; Use string functions to construct fid and exec (omitted).
      ;; Distribute:
       (dscpr_sub fid "loi" "dsc_lisp.build" exec)
      ) ; End loop
      ;;; Wait on all
      (dscpr_wait -1)
      ;;; Loop to construct file names for and open ".d" files
      ;;; for all subtasks, read reported factorizations,
      ;;; and perform a factor refinement ... (omitted)
    )) ; End let/cantorz
```

Figure 4: Code for distributed polynomial factorization.

	# parallel subtasks				
Digits	1	2	4	8	16
22	0:52(2)	0:54(5)	0:36(3)	1:22(5)	1:18(2)
	0:45(2)	0:54(5)	0:44(3)	1:00(3)	2:17(3)
	0:50(2)	0:54(5)	0:46(3)	1:08(3)	1:12(3)
	0:52(2)	0:54(5)	0:47(3)	0:51(3)	3:08(5)
43	31:37(7)*	7:43(6)	1:27(5)	3:07(9)	7:09(7)
	31:08(7)*	7:21(7)	2:40(8)	3:31(6)	7:47(9)
	33:39(7)*	7:20(7)	2:39(8)	2:25(7)	6:03(11)
		7:34(7)	2:46(8)	2:55(8)	4:13(5)

Figure 5: Timings for GKA (first stage, first descent strategy).
Times shown are elapsed minutes:seconds (# of descents).
* First descent with 110th discriminant on list.

titioned into 8 sublists and distributed, -763 appears 10th on the sublist examined by a Sparc, which takes under 9 minutes to discover it each time. Of course, with 16 subtasks, -763 will appear 5th on some sublist, but, unfortunately, this sublist was given to a particularly slow Sparc. As a result, the race to find a descent was won by a DecStation, which needed about 13 minutes to discover -4883 as its 83rd discriminant. Interestingly, -4883 appears 1335th on the original list, so once again a deeply buried discriminant emerges after sufficiently many halvings of the original list. In contrast to 201 digit data, the data for the 401 digit shows

Digits	# parallel subtasks 2	4	8	16	32	64
201	$39{:}07^{sp}$	$14{:}11^{sp}$	8:42	13:06	7:03	
	$38{:}53^{sp}$	$14{:}08^{sp}$	8:39	13:06	7:09	
	$19{:}52^{dec}$	$6{:}55^{dec}$	8:39	13:15	7:02	
	$19{:}47^{dec}$	$6{:}55^{dec}$	8:37	13:01	7:43	
401*			9:15	5:12	4:36	3:05

Figure 6: Timings for GKA, one descent only.
Times shown are elapsed minutes:seconds
* Times shown for 401-digit number are elapsed hours:minutes
^{sp} Denotes only Sparcstations used.
^{dec} Denotes only Decstations used.

Digits	# parallel subtasks 4	8	16	32
22	0:31(2)	0:50(3)	0:54(2)	2:02(2)
	0:30(2)	0:42(2)	0:59(2)	1:48(2)
43	4:18(6)	5:12(6)	5:07(5)	5:38(4)
	5:46(6)	4:40(6)	4:59(5)	8:30(5)

Figure 7: Timings for GKA (first stage, steepest descent strategy).
Times shown are elapsed minutes:seconds (# of descents).

# parallel subtasks 2	4	8	16	32
0:06	0:06	0:11	0:21	0:43
0:06	0:06	0:10	0:17	0:35
0:05	0:07	0:10	0:18	0:41
0:05	0:07	0:10	0:21	0:35

Figure 8: Timings for DSC overhead.
Times shown are elapsed minutes:seconds

Digits	Divide-and-Conquer	Distribute-and-Refine
26	0:53	0:13
36	3:14	1:43
46	9:18	2:41
56	17:23	6:56
66	24:10	8:46
76	35:47	11:30

Figure 9: Timings for Cantor-Zassenhaus factorization.
Times shown are elapsed hours:minutes.

steadily decreasing runtimes as the number of processors increases. Ideally, for very large primes, the goal is to have as many processors as there are discriminants, so that each processor need only examine one discriminant.

Figure 7 contains running times associated with the steepest descent strategy (50% completion) as outlined in §4.1. Observe that this strategy generally produces shorter first stage certificates but not necessarily faster running times.

Finally, our timings for the Cantor-Zassenhaus factorizations are shown in Figure 9. The traditional "divide-and-conquer" approach used a single processor, while the new "distribute-and-refine" approach outlined in §4.1 used 12 to 15 processors. The input to each was a d digit prime p, and a polynomial of degree $d-1$ over GF(p) which we computed as a product of $d-1$ random linear factors. The results show the new approach performing up to four times faster than its traditional counterpart.

5. Future Directions

The DSC system is designed to provide a general-purpose platform for distributing a symbolic computation. DSC implements the message passing paradigm for coarse grain parallel MIMD computation rather than a shared memory parallel computer. Such an approach has only become feasible with the introduction of implicit representation models for symbolic objects such as the black box representation for multivariate rational functions, that are extremely concise and yield a small packet size for distribution. Since DSC is still evolving, the system lacks features that we envision to be present in its final version.

There is a fairly substantial list of test problems whose computational solution can be speeded using a distributed approach. In particular, the problem of black box sparse multivariate polynomial interpolation and the problem of black box multivariate polynomial factorization is planned to be implemented with DSC.

Finally, we intend to provide a higher level to the user interface. For one, we hope to provide an interface between DSC and a standard computer algebra system such as Maple or Mathematica. Furthermore, several high level data types controlling the distribution and completion strategies of "bags" of parallel subtasks will be implemented and tested. It is also not unthinkable that DSC could eventually inspect sequential Lisp expressions for parallelizing breakup, such as in sufficiently compute-intensive `mapcar` constructs.

Literature Cited

Atkin, A. O. L. and Morain, F., "Elliptic curves and primality proving," *Math. Comput.*, p. submitted (September 1990).

Bach, E., Driscoll, J., and Shallit, J., "Factor refinement," in *Proc. ACM-SIAM Symp. Discrete Algorithms*; SIAM, Philadelphia, PA, pp. 201–211, 1990.

Cantor, D. G. and Zassenhaus, H., "A new algorithm for factoring polynomials over finite fields," *Math. Comp.* **36**, pp. 587–592 (1981).

Char, B. W., "Progress report on a system for general-purpose parallel symbolic algebraic computation," in *Proc. 1990 Internat. Symp. Symbolic Algebraic Comput.*, edited by S. Watanabe and M. Nagata; ACM Press, pp. 96–103, 1990.

Collins, G. E., Johnson, J. R., and Kuechlin, W., "PARSAC-2: A multi-threaded system for symbolic and algebraic computation," *Tech. Report* **TR38**, Comput. and Information Sci. Research Center, Ohio State University, December 1990.

Hitz, M., "HIPREC: Long integer arithmetic for SUN workstations and for the CRAY X/MP," *Master's Project*, Dept. Comput. Sci., Rensselaer Polytechnic Institute, Troy, N.Y., December 1988.

Johnson, J. R., "Designing Algebraic Algorithms for the Cray X-MP," *M.S. Thesis*, University of Delaware, January 1988.

Kaltofen, E., Lakshman, Y. N., and Wiley, J. M., "Modular rational sparse multivariate polynomial interpolation," in *Proc. 1990 Internat. Symp. Symbolic Algebraic Comput.*, edited by S. Watanabe and M. Nagata; ACM Press, pp. 135–139, 1990.

Kaltofen, E. and Trager, B., "Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators," *J. Symbolic Comput.* **9**/3, pp. 301–320 (1990).

Kaltofen, E., Valente, T., and Yui, N., "An improved Las Vegas primality test," *Proc. ACM-SIGSAM 1989 Internat. Symp. Symbolic Algebraic Comput.*, pp. 26–33 (1989).

Lenstra, A. K. and Manasse, M. S., "Factoring by electronic mail," *Proc. Eurocrypt '89, Springer Lect. Notes Comput. Sci.* **434**, (1989).

Melenk, H. and Neun, W., "REDUCE user's guide for the CRAY 1 / CRAY X-MP series running COS," *Tech. Report*, K. Zuse Zentrum Berlin, September 1986.

Ponder, C. G., "Evaluation of 'Performance Enhancements' in Algebraic Manipulation Systems," *Ph.D. Thesis*, Comput. Sci. Div. (EECS Dept.), Univ. California at Berkeley, 1988.

Roch, J.-L., "Calcul Formel et Parallélism," *Doctorate Thesis*, Inst. National Polytechnique, Grenoble, France, December 1989. In French.

Roch, J.-L., Senechaud, P., Siebert-Roch, F., and Villard, G., "Computer Algebra on a MIMD Machine," *Proc. ISSAC '88, Springer Lec. Notes Comput. Sci.* **358**, pp. 423–439 (1988).

Saunders, B. D., Lee, H. R., and Abdali, S. K., "A parallel implementation of the cylindrical algebraic decomposition algorithm," *Proc. ACM-SIGSAM 1989 Internat. Symp. Symbolic Algebraic Comput.*, pp. 298–307 (1989).

Seitz, S., "Verteiltes Rechnen in SAC-2," *Doctoral Dissertation*, Univ. Tübingen, June 1990. In German.

Silverman, R. D. and Stuart, S. J., "A distributed batching system for parallel processing," *Software—Practice Experience* **19**/12, pp. 1163–1174 (1989).

Sullivan, M. and Anderson, D., "Marionette: a system for parallel distributed programming using a master/slave model," in *Proc. 9th Internat. Conf. Distr. Comput. Syst.*; IEEE, pp. 181–188, 1989.

Watt, S. M., "Bounded Parallelism in Computer Algebra," *Ph.D. Thesis*, Dept. Comput. Sci., Univ. Waterloo, May 1986.