A DISTRIBUTED APPROACH TO PROBLEM SOLVING IN MAPLE

K. C. Chan, A. Díaz, and E. Kaltofen
Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY

Abstract

A system is described whereby a Maple computation can be distributed across a network of computers running Unix. The distribution is based on the DSC system, which can ship source code and input data to carefully selected computers for execution and which can retrieve the produced output data. Our code is fully portable and requires no changes of the underlying Maple or Unix systems. Speedup over Maple's built-in sequential procedure is demonstrated when computing determinants of integer matrices.

1. Introduction

The paradigm of distributing a compute-intensive program over a network of computers is becoming commonplace in today's problem solving by computer. Several systems are available for support of distributed computation, among them Gelernter's Piranha Linda, PVM (Parallel Virtual Machine) by Geist et al., and the Unix system's remote shell command. In 1991, we announced a platform for distributing symbolic code, called the Distributed Symbolic Computation tool—DSC (Díaz et al. 1991). The DSC system has been significantly improved over the past three years (Díaz et al. 1993). This paper describes our new interface to Maple, which permits the automatic distribution of a Maple computation over a network of computers.

The following important DSC features are available to Maple users through the new interface.

— The distribution of so-called parallel subtasks is performed from the Maple environment by a system call to a DSC program which communicates with the concurrent server daemon process. That process, which has established IP/TCP/UDP connections to equivalent daemon processes on the participating compute nodes, handles the call and sends the subtask to one of them. Similarly, the control flow of the application program is synchronized by library calls that wait for the completion of one or all subtasks. Participating compute nodes can be anywhere on the Internet.

— DSC can distribute a Maple source file and the corresponding input data file. The remote computer starts a Maple shell and executes the source file, which is assumed to read from the input data file and produce an output data file, which is returned to the parent process. Note that the distribution of source code, which is not restricted to Maple but can be in C or Lisp instead, allows the parent process to dynamically construct programs for distribution thus permitting the use of so-called "black box" data types. Furthermore, on a shared file system file transfer can be performed by path name and no physical file movement needs to take place.

— The master-slave paradigm for distribution can be relaxed by making use of a co-routine-like distribution mechanism. In that case, the parallel subtask can exchange information with the parent task in the middle of a computation.

— The interface from Maple to DSC consists of 10 library functions. Processor allocation and interprocess communication is completely hidden from the Maple programmer. Indeed, DSC has a fairly sophisticated scheduler that tries to match the subtask's resource demands, which are given in rough estimates as (optional) arguments to the DSC call, with a suitable com-

Appears in *Maple V: Mathematics and Application*, Proc. Maple Summer Workshop and Symp. '94, R. Lopez (ed.), pp. 13–21, Birkhäuser Verlag, Boston, 1994.

puter on the network. CPU and memory usage of participating compute nodes is estimated by having a resident daemon process probe them in 10 minute intervals and communicate the CPU and memory load to the DSC server daemons. If no computer meets certain threshold requirements, the parallel subtask gets queued for later distribution under presumably better load conditions on the network. The scheduler makes DSC a truly heterogeneous parallel system.

— The progress of a distributed computation can be monitored by an independently run controller program. This controller also initializes the DSC environment by establishing server daemons on the participating computers.

— It is possible to run several distributing Maple programs simultaneously by specifying distinct UDP port numbers on start-up. DSC and the Maple interface do not require any changes to existing Unix or Maple setups. Security is guaranteed because DSC tags the messages which it sends through unprotected ports with a secret key.

Note that the design of the features discussed above has been extensively tested in our symbolic applications, which are: the parallel Cantor/Zassenhaus polynomial factorization algorithm (Díaz et al. 1991), the Goldwasser-Kilian/Atkin primality prover for titanic primes, i.e., prime numbers with more than 1000 decimal digits (Valente 1992), and the block Wiedemann algorithm for the solution of sparse linear systems over finite fields (Díaz et al. 1993).

The main design goal for the DSC/Maple interface was that it had to be completely portable. Therefore, we chose a mechanism by which the DSC library functions are called through a system call. That call executes an interface program that sends a signal to a concurrent daemon process. It is that single daemon process which calls the C language DSC user library function which in turn communicates with the server process via a user datagram. Thus we avoid any dependence on calls to functions written in C from within Maple, a feature that is lacking in older Maple installations.

We demonstrate the usage of our interface by a standard example, namely the computation of the determinant of an integer matrix by homomorphic imaging and Chinese remaindering (McClellan 1973). We can report that determinants of 100×100 integer matrices with single digit entries are computed faster by distribution. We note that all of the needed computer algebra technology, such as finding prime numbers, computing matrix determinants modulo prime numbers, and the Chinese remainder algorithm, is provided by Maple. There are more sophisticated coarse grain parallel algorithms, such as root finding or sparse interpolation (Char 1990), which would require more custom-made Maple programming. In our opinion, an important application of the DSC interface is the parallelization of our Maple code for the solution of sparse linear systems. This code was developed as a prototype for our superfast C++ implementation (Kaltofen and Lobo 1994). Nonetheless, the distributed version of our prototype code would make parallel sparse linear system solving available to the entire Maple world.

We hope to add several new features to DSC and the Maple interface in the future. First, we are planning to build a graphical user interface to the control program which monitors all DSC processes and computers. Second, we are planning to implement some form of process migration. Although our scheduler attempts to find the best compute node for a given parallel subtask, often the choice is not optimal because unexpected load levels appear later at the selected computer. In that case, it would be very helpful if a partially completed task could be moved to another node. And finally, the Maple interface could be enhanced by high level data types that hide the actual explicit DSC calls, such as the "bags of incomplete futures" that we have implemented in our Lisp interface (Díaz et al. 1991).

2. System Layers

In Díaz et al. 1991 we described how conceptually the DSC system itself is organized in a multi-layered fashion, each layer drawing from its predecessor. Referring to Figure 1, the bottom layer consists of the interprocess 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, the new Maple/C interface and C user programs that use only the seven basic C library functions. The third layer draws on the Lisp/C interface and the Maple/C interface. This layer contains the Lisp library, basic Lisp functions and basic Maple functions. At the topmost layer lies the implementation of high level

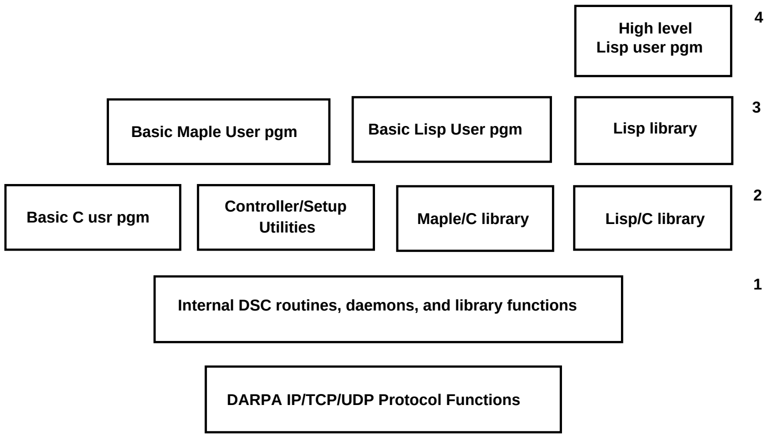

Figure 1: System Layers

Lisp functions which utilize the support routines contained in the Lisp library. Such a Library is being developed for Maple.

3. Maple Interface

The Maple User Interface Library contains ten functions that can be invoked from a user's Maple program or session. The standard example of computing the determinant of an integer matrix in parallel (by homomorphic imaging and Chinese remaindering) illustrates the interaction between DSC and Maple.

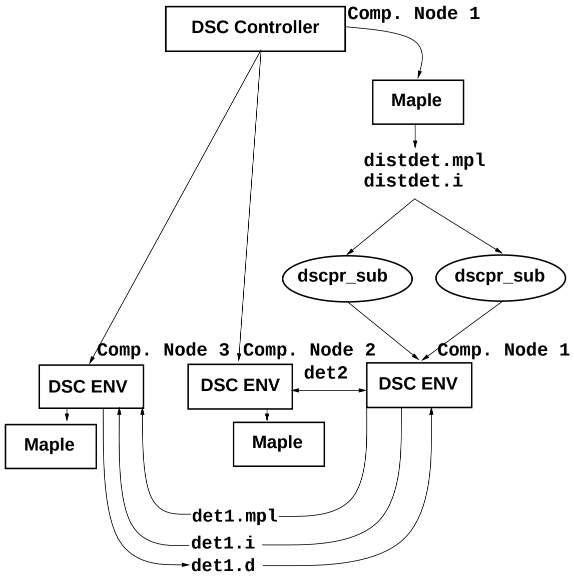

Figure 2: Simple Distribution Scenario

Before a Maple application can distribute parallel subtasks over the network, the database of available compute nodes needs to be initialized and DSC server daemons have to be started. The DSC Controller program supports the user in configuring the network and in the monitoring of computational tasks (Díaz et al. 1991).

Figure 2 portrays the Scenario of a distributed Maple determinant computation. After starting the controller program on compute node 1, and DSC servers on compute nodes 1, 2, and 3, the main Maple task `distdet.mpl` is initiated. It first reads in an integer matrix from `distdet.i` and selects appropriate moduli. For each modular image, `distdet.mpl` calls the DSC Maple library function `dscpr_sub`, passing Maple code in a file `det1.mpl` and the residue matrix in `det1.i` to DSC for distribution. The function `dscpr_sub` communicates the distribution request to the local server running on compute node 1. The local server locates a suitable processor, in this case compute node 3, and sends both files to the corresponding remote server. The server on compute node 3 starts a Maple session for `det1.mpl`. That program then reads its data from the remote copy of `det1.i` and writes its output to the remote file `det1.d`. When the parallel subtask completes the file, `det1.d` is copied back to compute node 1. In the meantime, the application program has initiated distribution of another parallel subtask, `det2`, for which the local server selected compute node 2. A call to the DSC Maple library function `dscpr_wait` or `dscpr_next` can be used to block the main task `distdet.mpl` on the completion of its children.

The source code of `distdet.mpl` found in Figures 3 through 9 illustrates the usage of the DSC Maple library functions.

```
with(linalg);
################################################
## distdet(subprobsf,solfile,A,iprod)
## input -
##          subprobsf - source file (without path information)
##                      of the subproblems
##          solfile   - the file name that will contain the determinant
##                       of the matrix A
##          A         - square matrix
##          iprob     - starting point for prime moduli
## description -
##          compute the determinant of the integer matrix A by
##          homomorphic imaging and Chinese remaindering
distdet := proc(subprobsf,solfile,A,iprod)
 local LOW_CPU, MEM, PWD, SOURCE_FILE,INDEX,fid,build,exec,y,
       NUMPROBS,moduli,residue,tmod,ubound,prod,inc,u;
 ## LOW_CPU     - estimated CPU usage
 ## MEM         - estimated memory usage in megabytes
 ## PWD         - real path
 ## SOURCE_FILE - source file with real path information
 ## INDEX       - subproblem index file
 ## NUMPROBS    - the number of primes needed given the initial value
 ## fid         - temporary file id
 ## build       - Maple build command
 ## exec        - Maple execute command
 ## y           - loop counter
 ## moduli      - the array of moduli
 ## residue     - the array of determinant residues
 ## tmod        - moduli for each subroblem (save can only save names)
 ## ubound      - the upper bound for the product of the moduli
 ## prod        - the product of the moduli
 ## inc         - the next starting point for the moduli search
 ## u           - determinant
 # Read in DSC Maple User Functions
 read(`dsc_maple.mpl`);
 # Set problem parameters
 subprobsf:= subprobsf;
 LOW_CPU:= 240;
 MEM:= 5;
 # Start dsc_maple daemon
 system(`dsc_maple commandfile messagepid dscmaplepid &`);
 # Wait a while for startup
 system(`sleep 2`);
 # Get real path information
 PWD := dscpr_pwd(`commandfile`,`messagepid`,`dscmaplepid`,`answerfile`);
 SOURCE_FILE:= cat(PWD,`/`,subprobsf);
 # Log problem in local server problem database
 INDEX := dscdbg_start(`commandfile`,`messagepid`,`dscmaplepid`,PWD,
                      `answerfile`);
```

Figure 3: Interface Start-Up

Figure 3 displays the initialization steps that are needed when distributing Maple code. Once the `dsc_maple.mpl` interface library has been loaded, a dsc_maple daemon can be started via a system call to the host operating system from the Maple environment. After startup of the dsc_maple daemon, calls to the functions in the Maple User Interface Library can be made. In order to assure distinct file names when running multiple distributed Maple applications, the dsc_maple daemon program must receive three command line arguments, which are file names representing a dsc_maple command file, a dsc_maple_message temporary daemon process id file, and a dsc_maple daemon process id file. These files, along with a fourth string denoting the dsc_maple_message answer file, are passed as parameters to all of the subsequent Maple User Interface Library functions. These four files are only used by the different layers of the Maple Interface and are never accessed directly from the user's Maple application. The function `dscpr_pwd` is used to determine the current path of the local Maple session. This information is used later as a parameter to all remaining Maple User Interface Library functions. The function `dscdbg_start` in Figure 3 is the first maple interface call that communicates a request to the local DSC server. This function can be used to track a problem and is useful when one wishes to debug distributed tasks using Maple interactively. Arguments to this function include the four interface specific file names and the current path of the local Maple session.

```
ubound := evalf(2*hadamard(A));

 # Compute the Moduli
 print('Compute_the_Moduli');
 prod := 1; NUMPROBS:= 0;
 inc :=iprod;

 while prod < ubound do
  NUMPROBS := NUMPROBS+1;
  moduli[NUMPROBS] := nextprime(inc);
  inc := moduli[NUMPROBS]+1;
  prod := prod * moduli[NUMPROBS];
 od;
```

Figure 4: Compute Necessary Moduli

One way of computing the determinant of a matrix in a distributed fashion is to first compute the determinant of the input matrix modulo n relatively prime numbers in parallel and then to construct the corresponding true integer value by Chinese remaindering. Figure 4 shows the computation of the moduli by using the Hadamard inequality to get an upper bound for the determinant.

```
# Prepare corresponding input files
 for y from 1 to NUMPROBS do
 fid:=cat(SOURCE_FILE,y,`.`,`i`);
 tmod := moduli[y];
 save A , tmod, fid;
 od;

 # Make copies of the subproblem
 copyfiles(NUMPROBS,SOURCE_FILE,`mpl`);
```

Figure 5: Prepare Parallel Subtasks

After the moduli have been computed, the parallel subtasks can be prepared for distribution. Figure 5 details how the parallel subtask input files and copies of the parallel subtasks are created. The `copyfiles` function is provided in the Maple User Functions Library. Eventually, we hope to provide additional high level library functions.

Figure 6 demonstrates the use of the function `dscpr_sub` to start a parallel subtask within the DSC environment. Additional arguments to this function are four strings denoting, respectively, the path to the local files, the types of the local files to be sent, the build (compile, for instance) command, and the exec (load and run) command required for the parallel subtasks; two additional integer arguments represent the CPU and memory requirements of 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 C and Lisp compilation by supplying `dsc_c.build` or `dsc_lisp.build` as the build command. The shell script `dsc_maple.int` applies the source and input files to a Maple session.

After the determinant parallel subtasks have been spawned Figure 7, shows the Maple code necessary for blocking the execution of the main task and after synchonization the code required for performing the shutdown of the DSC Maple interface. The functions `dscpr_wait` and `dscpr_kill` are used, respectively, to wait on and to kill specific subtasks. In both cases, the additional argument is the index of the subtask in question. The `dscpr_wait` 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 additional argument is filled with the name of the solution file corresponding to the completed subtask. The function `dscpr_maple_kill`, will terminate the dsc_maple daemon in an orderly fashion.

```
for y from 1 to NUMPROBS do
    # Prepare arguments
    fid:= cat(SOURCE_FILE,y);
    build:= cat(`dsc_maple.build`);
    exec:= cat(`dsc_maple.int `,subprobsf,y,`.mpl `,subprobsf,y,`.d`,
              ` `,subprobsf,y,`.i`);
    print(`one_level_driver: fid = `.fid);
    print(`one_level_driver: build = `.build);
    print(`one_level_driver: exec = `.exec);
    print(`one_level_driver: cpu requirement = `.LOW_CPU);
    print(`one_level_driver: mem requirement = `.MEM);

    # Spawn subproblems
    rval:= dscpr_sub(`commandfile`,`messagepid`,`dscmaplepid`,
                     PWD,fid,`pi`,build,exec,LOW_CPU,MEM,
                    `answerfile`);
od;
```

Figure 6: Distribute Parallel Subtasks

```
    # Wait on all subproblems
    rval := dscpr_wait(`commandfile`,`messagepid`,`dscmaplepid`,
                       PWD,-1,`answerfile`);
dscpr_maple_kill(`commandfile`,`messagepid`,`dscmaplepid`,PWD);
```

Figure 7: Wait On All Parallel Subtasks

```
 # get residues from .d files
 for y from 1 to NUMPROBS do
  fid:= cat(SOURCE_FILE,y,`.d`);
  read(fid);
  residue[y] := ddet;
 od;
 u := chrem(convert(residue,list),
           convert(moduli,list));
 save u , prod ,solfile;
 RETURN(u);
end;
```

Figure 8: Compute Determinant

The code in Figure 8 computes the integer determinant of A from the n residues of the determinants by Chinese remaindering.

```
read(`IM_10_10_1`);
answer := distdet(`det`,`distdet.d`,
                  A,10000000);
quit;
```

Figure 9: Initial Startup Mechanism

Since the file is executed in a interpreted manner the call to `distdet` is made after the function definition and the reading of the matrix. Finally the session is terminated via a `quit` command as seen in Figure 9.

Figure 10 is the subproblem file copied and distributed by `distdet.mpl` which computes the determinant of the integer matrix by homomorphic imaging.

The input file and output file specified by the call to `dscpr_sub` in the main task are put in the variables DSC_IFILE and DSC_OFILE by the Maple interface.

The control program allows the user to inspect the progress of an application program and its parallel subtasks. This will include the main application program and the parallel subtasks that have been spawned in the current DSC environment.

Figure 11 is an example of the parallel subtask database. The main Maple program prefixed as `distdet` (running on `troi.cs.rpi.edu`) registered with the DSC server via `dscdbg_start`, was given the task name `DSC40e0_1`. This task distributed Maple code (and an input file) to `pleiades.cs.rpi.edu`. This is indicated by both `maple: 1` and `input: 1` being set. The shell script `dsc_maple.int` is in the exec argument of the call to `dscpr_sub`. The parallel subtask has not spawned children. All processes run on Sun compute nodes, as `CPU 3` indicates.

For 100×100 integer matrices with single digit entries the computation of the the distributed version of the determinant computation program on 22 processors was 1.5 times faster than the standard Maple determinant function (det). For 200×200 matrices on 18 processors the speed up was a factor of 1.9. The speed up becomes more

```
with(linalg);
################################################
## ddet(one_input,one_output)
## input -
##        one_input - input file for subproblem
##        one_output - output file for subproblem
ddet := proc(one_input,one_output)
 local commandfile,messagepid,dscmaplepid,answerfile,msgstr,PWD;
 ## commandfile  - dsc_maple DSC command file
 ## messagepid   - dsc_maple_message process id file
 ## dscmaplepid  - dsc_maple process id file
 ## answerfile   - subtask solution file
 ## msgstr       - temporary string
 ## PWD          - real path
 # Read in DSC Maple User Functions
 read(`dsc_maple.mpl`);
 # Start dsc_maple daemon
 commandfile := cat(one_input,`.cmd`);
 messagepid  := cat(one_input,`.mpid`);
 dscmaplepid := cat(one_input,`.ppid`);
 answerfile  := cat(one_input,`.ans`);
 msgstr := cat(`dsc_maple `,commandfile,` `,messagepid,` `,dscmaplepid,` &`);
 print(msgstr);
 system(msgstr);
 # Wait a while for startup
 system(`sleep 2`);
 # Get real path information
 PWD := dscpr_pwd(commandfile,messagepid,dscmaplepid,answerfile);
 # Kill dsc_maple_daemon
 dscpr_maple_kill(commandfile,messagepid,dscmaplepid,PWD);
 # Wait a while for shut down
 system(`sleep 2`);
 # Remove files
 msgstr := `rm -f `; msgstr := cat(msgstr,commandfile); system(msgstr);
 msgstr := `rm -f `; msgstr := cat(msgstr,messagepid);  system(msgstr);
 msgstr := `rm -f `; msgstr := cat(msgstr,dscmaplepid); system(msgstr);
 msgstr := `rm -f `; msgstr := cat(msgstr,answerfile);  system(msgstr);
 # Read in matrix and modulo
 msgstr := PWD;
 msgstr := cat(msgstr,`/`,one_input);
 read(msgstr);
 # compute the determinant of A modulo tmod
 ddet := Det(A) mod tmod;
 save ddet , one_output;
end;
ddet(DSC_IFILE,DSC_OFILE);
quit;
```

Figure 10: Parallel Subtask

significant with larger problems.

The Maple interface library also allows the user to implement co-routines. To distribute Maple co-routine applications, the function `dscpr_cosetup` must be called at the beginning of any parallel subtask that is to be treated as a co-routine. Its only arguments are the four interface specific file names and the current path of the local Maple ses-

```
P R O B L E M   D A T A B A S E

 problem name: distdet
 problem  PID:  16596   node: troi.cs.rpi.edu   index   0       CPU  3
 parent   PID:  16477   naddr: 128.213.2.43     index  -1
 source  name: /fs5/misc1/ugrads/chank/dsc/source/distdet
            c: 0  lisp: 0  math: 0 maple: 1 obj: 0  input:0
build command: dsc_maple.build
 exec command: dsc_maple.int /fs5/misc1/ugrads/chank/dsc/source/distdet.mpl
                            /fs5/misc1/ugrads/chank/dsc/source/distdet.d
 blocked sock:  -1       block condition -1
 completion  :   0      status: 00000000 build:   1
 peer link   :  -1      child list:   -1        # subprob:   0
 state       :   0
 estimated cpu use : LOW_CPU     estimated mem need (megs) : 5
MORE
 problem name: DSC40e0_1
 problem  PID:  16608   node: troi.cs.rpi.edu   index   1       CPU  3
 parent   PID:     -1   naddr: 128.213.2.43     index  -1
 source  name: dsc_prob_debug
            c: 0  lisp: 0  math: 0 maple: 0 obj: 0  input:0
build command:
 exec command:
 blocked sock:   6       block condition -1
 completion  :   0      status: 00000000 build:   1
 peer link   :  -1      child list:    2        # subprob:   2
 state       :   0
 estimated cpu use : -1 (UNKOWN VALUE)
 estimated mem need (megs) : -1
MORE
 problem name: DSC40e02
 problem  PID:     -1   node: pleiades.cs.rpi.edu       index   2       CPU  3
 parent   PID:  16608   naddr: 128.213.2.43     index   1
 source  name: /fs5/misc1/ugrads/chank/dsc/source/det1
            c: 0  lisp: 0  math: 0 maple: 1 obj: 0  input:1
build command: dsc_maple.build
 exec command: dsc_maple.int det1.mpl det1.d det1.i
 blocked sock:  -1       block condition -1
 completion  :   1      status: 00000000 build:   0
 peer link   :   3      child list:   -1        # subprob:   0
 state       :   0
 estimated cpu use : HIGH_CPU    estimated mem need (megs) : 5
MORE
```

Figure 11: Problem Data Base

sion. When the Maple parallel subtask calls the `dscpr_cowait` function with an additional integer argument, it enters a sleep state and optionally transmits a data file back to its parent. A parent task can send a wake up call to a sleeping parallel subtask via the `dscpr_coresume` function. Additional arguments to this function are an integer which uniquely identifies a parallel subtask (returned from the spawning call to `dscpr_sub`) and a string identifying which input file, if any, should be sent to the co-routine before the parallel subtask is to be resumed.

This material is based on work supported in part by the National Science Foundation under Grant No. CCR-90-06077 and Grant No. CCR-93-

19776, Research Experiences for an Undergraduate supplement (first author), and by GTE under a Graduate Computer Science Fellowship (second author).

The software described in this paper is freely available by anonymous ftp from ftp.cs.rpi.edu in directory dsc.

Literature Cited

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.

Díaz, A., "DSC Users Manual (2nd ed.)," *Tech. Rep.* **93-11**, Dept. Comput. Sci., Rensselaer Polytech. Inst., Troy, New York, May 1993. 197 pp.

Díaz, A., Hitz, M., Kaltofen, E., Lobo, A., and Valente, T., "Process scheduling in DSC and the large sparse linear systems challenge," in *Proc. DISCO '93*, Springer Lect. Notes Comput. Sci. **722**, edited by A. Miola; pp. 66–80, 1993. Available from `anonymous@ftp.cs.rpi.edu` in directory `kaltofen`.

Díaz, A., Kaltofen, E., Schmitz, K., and Valente, T., "DSC A System for Distributed Symbolic Computation," in *Proc. 1991 Internat. Symp. Symbolic Algebraic Comput.*, edited by S. M. Watt; ACM Press, pp. 323-332, 1991. Available from `anonymous@ftp.cs.rpi.edu` in directory `kaltofen`.

Kaltofen, E. and Lobo, A., "Factoring high-degree polynomials by the black box Berlekamp algorithm," *Manuscript*, January 1994. Available from `anonymous@ftp.cs.rpi.edu` in directory `kaltofen`.

McClellan, M. T., "The exact solution of systems of linear equations with polynomial coefficients," *J. ACM* **20**, pp. 563–588 (1973).

Valente, T., "A distributed approach to proving large numbers prime," *Ph.D. Thesis*, Dept. Comput. Sci., Rensselaer Polytech. Instit., Troy, New York, December 1992. Available from `anonymous@ftp.cs.rpi.edu` in directory `valente`.

King Choi Chan currently is an undergraduate student in Computer Systems Engineering at Rensselaer Polytechnic Institute.

Angel Díaz received both his B.S. degree in Computer in 1991 and his M.S. degree in Computer Science in 1993 from Rensselaer Polytechnic Institute. He is currently pursuing the Ph.D. degree at Rensselaer under the direction of Professor Erich Kaltofen. His special fields of interest include parallel methods in symbolic computation. He was a Rensselaer Polytechnic Institute Graduate Fellow from 1991–1992, received support from the National Science Foundation Fellowship Program in 1992, and is currently the recipient of the GTE Fellowship Program. He is also a member of Upsilon Pi Epsilon national computer science honor society.

Erich Kaltofen received both his M.S. degree in Computer Science in 1979 and his Ph.D. degree in Computer Science in 1982 from Rensselaer Polytechnic Institute. He was an Assistant Professor of Computer Science at the University of Toronto and an Assistant and Associate Professor at Rensselaer Polytechnic Institute, where he is now a Professor. His current interests are in computational algebra and number theory, design and analysis of sequential and parallel algorithms, and symbolic manipulation systems and languages. Professor Kaltofen currently is the Chair of ACM's Special Interest Group on Symbolic & Algebraic Manipulation and serves as associate editor on several journals on symbolic computation. From 1985–87 he held an IBM Faculty Development Award. From 1990–91 he was an ACM National Lecturer.

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Internet: {`diaza`,`kaltofen`}`@cs.rpi.edu`