J. Symbolic Computation (1995) **19**, 269–282

Process Scheduling in DSC and the Large Sparse Linear Systems Challenge[†]

A. DÍAZ, M. HITZ, E. KALTOFEN, A. LOBO AND T. VALENTE[‡]

Department of Computer Science, Rensselaer Polytechnic Institute

Troy, New York 12189-3590, U.S.A.

Internet: {diaza,hitzm,kaltofen,loboa}@cs.rpi.edu; tomv@tiger.hsc.edu

(Received 15 March 1995)

New features of our DSC system for distributing a symbolic computation task over a network of processors are described. A new scheduler sends parallel subtasks to those compute nodes that are best suited in handling the added load of CPU usage and memory. Furthermore, a subtask can communicate back to the process that spawned it by a co-routine style calling mechanism. Two large experiments are described in this improved setting. In the first we have implemented an algorithm that can prove a number of more than 1,000 decimal digits prime in about 2 months elapsed time on some 20 computers. In the second a parallel version of a sparse linear system solver is used to compute the solution of sparse linear systems over finite fields. We are able to find the solution of a 100,000 by 100,000 linear system with about 10.3 million non-zero entries over the Galois field with 2 elements using 3 computers in about 54 hours CPU time.

1. Introduction

In Díaz *et al.,* (1991) we introduced our DSC system for distributing large scale symbolic computations over a network of UNIX computers. There we discuss in detail the following features:

(i) The distribution of so-called parallel subtasks is performed in the application program by a DSC user library call. A daemon process, which has established Internet Protocol (IP) connections to equivalent daemon processes on the participating compute nodes by use of both the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), handles the call and sends the subtask to one of them.

[†] This material is based on work supported in part by the National Science Foundation under Grant No. CCR-90-06077, CDA-91-21465, and CDA-88-05910, and by GTE under a Graduate Computer Science Fellowship (A. Díaz). A preliminary version of this paper appeared in "Design and Implementation of Symbolic Computation Systems," A. Miola (ed.), *Springer Lect. Notes Comput. Science*, **722**, 66–80 (1993).

[‡] Valente's current address is Department of Mathematics and Computer Science, Hampden-Sydney College, Box 161, Hampden-Sydney, Virginia 23943, USA.

0747-7171/95/010269 + 14 $08.00/0 © 1995 Academic Press Limited

Similarly, the control flow of the application program is synchronized by library calls that wait for the completion of one or all subtasks.

(ii) DSC distributes not only remote procedure calls to precompiled programs, but also programs that are first compiled on the machine serving the subtask. This enables the distribution of dynamically generated "black-box" functions (cf. Kaltofen and Trager, 1990) and easy use of computers of different architecture.

(iii) DSC can be invoked from C, Common Lisp and Maple programs. It can distribute within a local area network (LAN) and across the Internet.

(iv) The interface to the application program consists of seven library functions. Processor allocation and interprocess communication is completely hidden from the user.

(v) 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.

(vi) We document experiments with DSC on a parallel version of the Cantor/Zassenhaus polynomial factorization algorithm and the Goldwasser-Kilian/Atkin (GKA) integer primality test.

New experiments with the GKA primality test that run on so-called "titanic" integers, i.e., integers with more than 1000 decimal digits, and experiments with a parallel sparse linear system solver, namely, Coppersmith's block Wiedemann algorithm (Coppersmith, 1994), have lead to several key modifications to DSC. In this article we describe these changes, as well as the results obtained by applying the improved environment to both titanic primality testing and sparse linear system solving.

Unlike on a massively parallel computer, where each processor has the same computing power and internal memory, a network of workstations and machines is a diverse computing environment. At the time the application program distributes a subtask, the DSC server has to determine which machine will receive this subtask. Our original design used a round-robin schedule, which resulted in unsatisfactory subtask-to-processor allocation. The new scheduler continuously receives the CPU load and memory usage of all participating machines, which are probed by resident daemon processes at 10 minute intervals. In addition, the application program supplies an estimate of the amount of memory and a rough measure of CPU usage. The scheduler then makes a sophisticated selection of which processor is to handle the subtask. If certain threshold values are not met, the subtask gets queued for later distribution under hopefully better load conditions on the network. The details of the scheduling algorithm are described in §2.1. Without this very fine tuned distribution scheduler, neither the primality tester nor the sparse linear system solver could have been run on inputs as large as the ones we had. Note that DSC's ability to account for the heterogeneity of the compute nodes is one feature that clearly distinguishes it from other parallel computer algebra systems such as Maple/Linda (Char, 1990), PARSAC-2 (Collins *et al.,* 1990), the distributed SAC-2 of Seitz, (1992), or PACLIB (Hong and Schreiner, 1993).

DSC supports a very coarse grain parallelism. This was quite successful for the primality tester, where each parallel subtask is extremely compute intensive and uses a moderate amount of memory. However, the Wiedemann sparse linear system solver can be implemented by slicing the coefficient matrix and storing each slice on a different processor. These slices will repeatedly be multiplied by a sequence of vectors. We have implemented a mechanism whereby the subtasks remain loaded in memory (or swap

space) on first return, and can be continued at the point following the previous return with different data supplied by the calling program, much like co-routines (see §2.3). This introduces a finer grain parallelism and allows two-way communication between the subtask and the parent process. This co-routine mechanism tends to make use of the distributed memory more than the parallel compute power.

DSC has also been modified internally in two important ways. First, the environment can now be initialized on a user supplied UDP port number. Several users can thus set up individual DSC servers without interfering with one another. We note that the inter-process communication does not take place on the system level, where a single port number could have been reserved for DSC. Hence no system modifications are necessary to run DSC, which is often desired when linking to off-site computers. Nonetheless, the port number is public and servers could be started, perhaps maliciously, to communicate with an existing environment. We guard against such mishap by tagging each message with a key set by the individual user. More details on these enhancements are found in §2.2.

Our first test problem has been the GKA primality test applied to numbers with more the 1000 decimal digits. We are successful in proving the primality of a 1111 digit number on a LAN of some 20 computers in about 2 months turnaround time. The details and observations of this experiment are described in §2.4. Our second test problem is a distributed version of the Coppersmith block Wiedemann algorithm. This algorithm for solving unstructured sparse linear systems has very coarse grain size, unlike classical methods such as conjugate gradient, which makes it very suitable for the DSC environment. We have implemented two variants, one for entries being from prime finite fields whose elements fit into 16 bits, and one for entries from GF(2), the field with two elements. In the latter case, we not only realize Coppersmith's processor internal parallelism by performing the bit operations simultaneously on 32 elements stored in a single computer word, but we further "doubly" block the method and distribute across the network. The details of our experiments are described in §3; we are successful in speeding up the solution of linear systems over GF(2) of more than 100,000 equations and variables with over 10 million non-zero coefficient matrix entries by factors of 3 and more. Such large runs would very likely not have completed using our old round-robin scheduling, since only the selection of compute nodes with large memory makes our programs feasible.

2. New Features of DSC

The goal of process scheduling in DSC is to locate available resources in the network and to distribute subtasks without creating peak loads on any node. Selection of a suitable computer is based on three factors: *rating of resources, requirements for subtask,* and *load on nodes.*

2.1. The Scheduler

The hardware resources are defined in three fields of the node list: the core memory size in MByte, the CPU power in MIPS and the number of processors. In the application program the user has to specify an estimate for the expected memory needs in MByte and a "fuzzy" value (LOW, MEDIUM, HIGH) for CPU usage.

In order to provide the scheduler with information about the current and the expected load at each compute node, a method of data collection had to be developed. The ideal

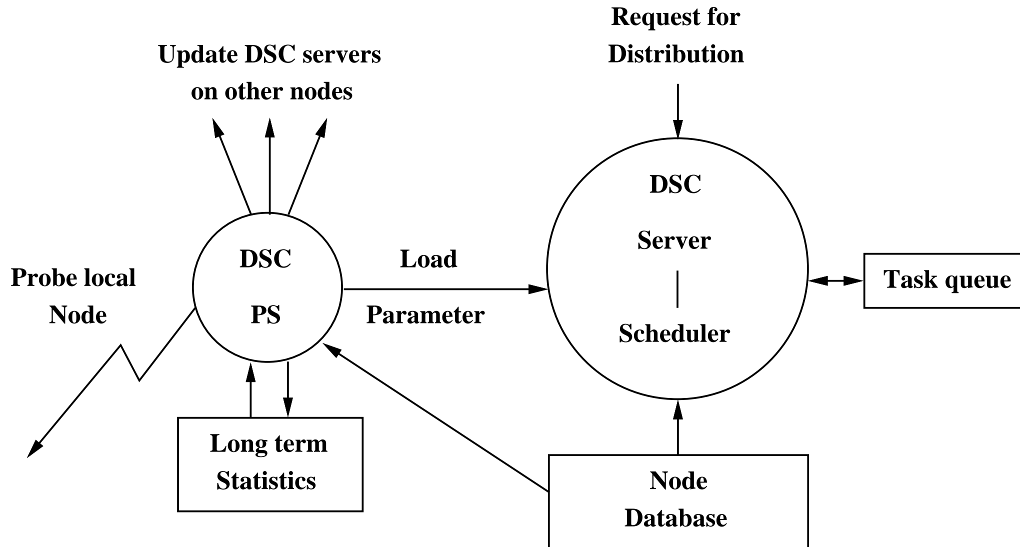

Figure 1: Interplay between DSC server and DSC_ps on a single compute node.

load-meter would provide exact data in real time with some corrections derived from trend predictions. This would require the monitoring program to be tied to the operating system on a low level. Unfortunately this would place an excessive burden on the user (request for higher privileges) and it would make the system less portable. However, most of the time it is not desirable to have measurements with high resolution. The readings should reflect trends for longer time periods rather than just be snapshots. As a first solution the UNIX `ps` command was chosen to measure CPU and memory usage about 8 to 10 times an hour. Due to the latency (up to one minute) involved with `ps`, and for better modularity, a separate process "DSC_ps" was added in the current version of DSC.

Once a DSC server is running, it spawns off the DSC_ps process for its node. DSC_ps maintains a table of statistics, which is saved to disk after each update. At the end of each hour, the mean values of CPU and memory usage are averaged with the previous values for the corresponding hour of the day. At the end of the day (or week) the values for corresponding day of the week (or week of the year) are adjusted by the latest readings. From all four levels (current reading, hour, day and week) a weighted average is computed to include long term effects. The resulting two values (CPU, memory) are sent to the local DSC server which in turn will communicate the update to all other servers on the network (see Figure 1). The backup file allows the initialization of load parameters according to anticipated patterns of usage. DSC_ps will then adapt those guessed values with respect to the new readings.

The scheduler in the DSC server uses the values received from DSC_ps, the ratings from the node database, and the estimated needs of the next task to select the target machine. For this purpose a sorted list of compute nodes which satisfy a minimum requirement of available memory is maintained. Based on the memory estimate of the application, all nodes which would stay above a certain threshold (allowing for some moderate paging) are preselected. Among them the one with the lowest CPU usage is finally chosen for

distribution. If none of the nodes can satisfy the requirements, the job is put back into a queue until the load on one of the computers decreases to a sufficient level.

After distributing the new task the server adjusts for the expected change in the load parameters of the selected node. Because of the long latency period, it cannot wait for the next readings of the actual load when it has to distribute many tasks in a short period of time. In time it can replace the estimates by the actual values whenever new load readings are received from the other server. This has the convenient side effect that the distributing server does not have to rely on transient measurements resulting from the startup phase of the new task (which can involve compilation of source code). Most of the time it will receive the steady-state readings because the server of the selected compute nodes will send the update of the load parameters with low priority.

2.2. Interprocess Communication and Message Validation

DSC uses the User Datagram Protocol (UDP) for most of its communication and Transmission Control Protocol (TCP) stream sockets for file transfer. For the sake of portability, all inter-process communication adheres to the DARPA Internet standard TCP/IP/UDP as implemented in UNIX 4.2/4.3bsd (see Díaz 1992). This low level approach avoids the high latency present in the UNIX `rsh` and `ftp` commands, and it provides real time information on subtasks and compute nodes for possible control actions such as subtask rescheduling. Before a user starts an application program that distributes parallel subtasks over the network, the DSC server daemons must be started. These daemon programs execute in the background and monitor a single UDP datagram address for new external stimuli from other DSC servers, the DSC controller program, the resource and work load monitor daemon program DSC_ps, and application programs. In order for a client process to contact a DSC server, the client must have a way of identifying the DSC server it wants. This can be accomplished by knowing the 32-bit Internet address of the host on which the server resides and the 16-bit port number which identifies the destination of the datagram on the host machine. Each DSC server must be using the same UDP port number in order to communicate with the others. UDP port numbers are not reserved and can be allocated by any process. The run time port number allocation option allows the user to automatically poll the machines in the configured DSC network to find a suitable port number for the initiation of a set of DSC servers. This is done by the DSC control program and consequently all DSC servers can be started using the determined available port number for their UDP communication.

If the control program could not establish a connection to a DSC server via the port number specified in the configuration file, the control program will assume that no active DSC server is monitoring this port. Consequently, the control program searches for an available port number which is not used on any machines in the "farm" of compute nodes in the DSC network. Optionally, the user may specify its own port number thereby bypassing the runtime port number allocation mechanism.

Once a port number has been determined the control program will start up the remote DSC servers via a `rsh` command supplying the executable and the port number to the remote or local compute node. Once all DSC servers are active communication takes place only via the IP/TCP/UDP protocols.

The primary function of the DSC server daemon program is to monitor a single UDP datagram address for incoming messages. Each message is a request for the DSC server to perform some action. However, in order to act only on messages received from the user

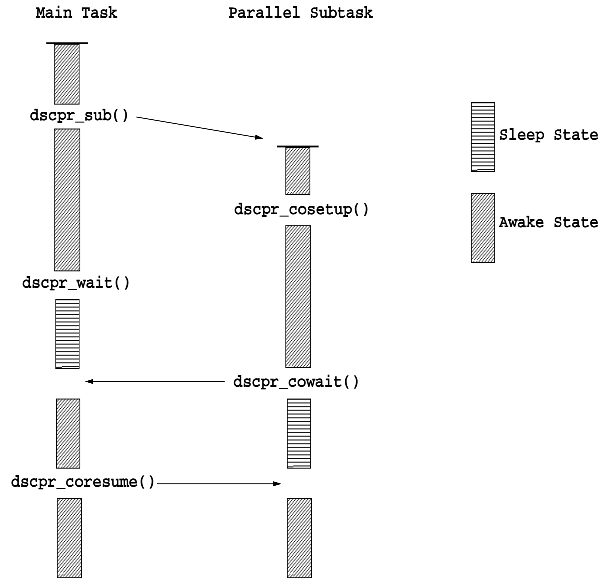

Figure 2: Co-routine flow of control.

that started the server, all messages contain a message validation tag which is specified by the user. If for any reason the message validation tag received by the DSC server does not match the server's message validation tag, the message is ignored and the invalid action request is logged. This avoids the inadvertent message passing that could occur when multiple DSC systems execute concurrently in the same open network computing environment using the same datagram port number.

2.3. Co-Routines

The C and Lisp DSC application programmer can take advantage of the resources found in the DSC network by utilizing 5 base functions callable from a user's program. The function `dscpr_sub` is used for the activation of parallel subtasks and designating their respective resource usage specifications. The calls to `dscpr_wait`, `dscpr_next`, and `dscpr_kill` are used to wait on a specific parallel subtask or on the completion of all parallel subtasks, to wait for the next completed parallel subtask and to kill a specific parallel subtask, respectively. Finally, the function `dscdbg_start` can be used to track a task and is useful when one wishes to debug tasks using interactive debuggers such as Unix's `dbx`. In order to meet the sparse linear system challenge (see §4), where there is a need to maintain large amounts of data within parallel subtasks, the C User Library has been extended to allow the user to implement co-routines (Kogge 1991, §9.6.3). The function `dscpr_cosetup` must be called at the beginning of any parallel subtask that is to be treated as a co-routine. This initialization is necessary so that the wake up signal received by a parallel subtask from the DSC server can be interpreted as a command to resume execution of the subproblem. Specifically, the `dscpr_cosetup`

function specifies how the subtask process will handle an asynchronous software interrupt by providing the address of an internal DSC function that wakes the process from a sleep state when the corresponding interrupt signal is detected. When the subtask calls the `dscpr_cowait` function, it enters a sleep state and optionally transmits a data file back to its parent. Once a co-routine parallel subtask or a set of co-routine parallel subtasks has been spawned by a call to `dscpr_sub`, the returned indices have been recorded, and a successful wait has completed, the parent task can send a wake up call to a sleeping parallel subtask via the `dscpr_coresume` function. Arguments to this function are an integer which uniquely identifies a parallel subtask (returned from the spawning call to `dscpr_sub`) and a string which identifies which input file if any should be sent to the co-routine before the parallel subtask is to be resumed. This call essentially generates the software interrupt needed for the waking of the sleeping parallel subtask. Figure 2 denotes the relationship that could exist between DSC utility function calls in a main task and its co-routine parallel subtask child.

2.4. The GKA Primality Test

In this section, we describe new experimental results with our distributed implementation of the Goldwasser-Kilian/Atkin (GKA) primality test (Atkin and Morain, 1993), which uses elliptic curves to prove an integer p prime; for earlier results, see Kaltofen *et al.,* 1989; Díaz *et al.,* 1991. In particular, we discuss here our success in proving "titanic" integers, i.e., integers with more than 1000 decimal digits, prime (see also Valente, 1992; Morain, 1991).

Let us briefly summarize the algorithm. The test has two phases: in the first phase, a sequence $\{p_i\}$ of probable primes is constructed, such that $p = p_0 > p_1 > p_2 > \cdots > p_n$. Each p_{i+1} is obtained from p_i by first finding a discriminant d such that p_i splits as $\pi\overline{\pi}$ in the ring of integers of the field $\mathbf{Q}(\sqrt{d})$. If $(1-\pi)(1-\overline{\pi})$ is divisible by a sufficiently large (probable) prime q, we set p_{i+1} to q, thus "descending" from p_i to p_{i+1}. We then repeat the process, seeking a descent from p_{i+1}. The first phase terminates with p_n having fewer than 10 digits. In the second phase, it is necessary to construct, for each p_i from the first phase, an appropriate elliptic curve over $\mathrm{GF}(p_i)$ which is used to prove p_i prime, provided p_{i+1} is prime. This results in a chain of implications

$$p_n \text{ prime } \Longrightarrow p_{n-1} \text{ prime } \Longrightarrow \cdots \Longrightarrow p_0 = p \text{ prime.}$$

In our experiment, we started with a probable prime number of 1111 decimal digits. Our code is written in the C programming language calling the Pari library functions (Batut *et al.,* 1991) for arbitrary precision integer arithmetic. Each time a descent is required in the first phase, a list of nearly 10,000 discriminants is examined. In fact, we chose to search all d with $|d| \leq 100{,}000$, where $\mathbf{Q}(\sqrt{d})$ has class number ≤ 50. Unfortunately, when p is titanic, few if any of these discriminants will induce a descent. For our prime of 1111 digits, we distributed the search for a descent from p_i to p_{i+1} to 24 subtasks, each of which is given approximately 400 discriminants to examine. The first subtask to find a descent reports it to the main task which then redistributes in order to find a descent from p_{i+1}. We required 204 descents before a prime of 10 digits was obtained. Our first phase run with the 1111 digit number as input began on January 12, 1992, and ended on February 13, 1992. The total elapsed time for the run was measured at about 569 hours, or approximately $3\frac{1}{2}$ weeks. Figure 3 depicts the progress of this

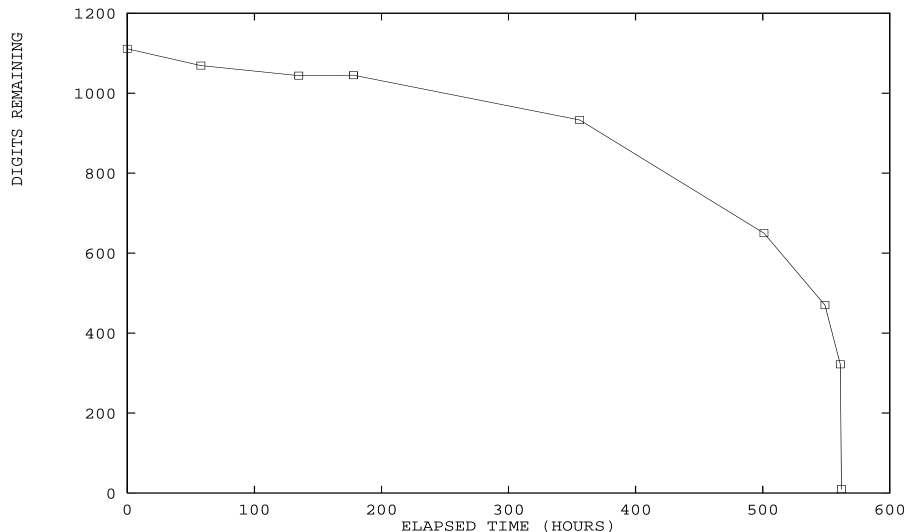

Figure 3: Graph of progress of GKA first phase.

run during this period. Notice that after 135 elapsed hours, the 1111 digit number is "reduced" to a number having 1044 digits. After an additional 43 hours, it appears that we regress, because the number shown now has 1045 digits! In fact, what has happened is that our program failed to find a descent from the 1044 digit number, and was forced to backtrack to a larger prime and find an alternate descent. Slow but steady progress is evident, until the last day, when the 322 digit number rapidly shrinks, and the first phase suddenly ends. Interestingly, it appears that about half of the total elapsed time of this run is spent merely reducing the original number to subtitanic size.

For our second phase run with 1111 digit inputs, there are a total of 204 descents to process. Typically, each of the 20 or so workstations is given about 10 descents. For each descent, the subtask must construct a class equation for the class field over $\mathbf{Q}(\sqrt{d}\,)$, find a root of the class equation, then use this root to construct an elliptic curve over the appropriate prime field GF(p). Once this curve is found, verification proceeds by finding a point on the curve which serves as a witness to the primality of p. Difficulties arise when the root-finder must handle class equations of degree 30 or more. Since the second phase is so sensitive to the degree of the class equation, it is critical that in the first phase we do whatever is possible to insure that only discriminants of relatively low class numbers are passed on to the second phase. Because of these factors, our phase two run for this titanic input takes approximately three weeks elapsed time. In a situation like this where a distributed subtask has a long running time it is much more difficult to schedule the task on a processor with expected low load, thus insuring high performance. In comparision to Morain's (1991) implementation, ours is competitive. Morain needs about 1 year of total SUN 3 CPU time for a 1065 digit number, using a different long integer arithmetic package.

3. Solving Large Sparse Linear Systems

We now report on the use of DSC in experiments with the Block Wiedemann Algorithm. Our distributed implementation follows Coppersmith's (1994) generalization of the sequential algorithm (Wiedemann, 1986). For the purposes of our paper we present a brief description below.

3.1. Background

The object is to find a non-trivial solution of a homogeneous system of linear equations with coefficient matrix denoted by B whose entries are from an abstract field K. The matrix is assumed to be square with dimension N, and it is supplied as a black box which accepts as its input a vector y and produces the vector By as its output. This task is called an application of the black box.

Our program returns one or more non-zero vectors w satisfying $Bw = 0$ if they exist. The vectors may not always be different. Internally, the program creates vector blocks $\mathbf{x} \in \mathsf{K}^{N\times m}$ and $\mathbf{z} \in \mathsf{K}^{N\times n}$ whose components are randomly selected from K. We refer to m and n as the row and column blocking factors and have set $m = n$ in our implementation and refer to them as n in the discussion below.

There are three stages to the algorithm. The first stage computes the polynomial $A(\lambda)$ of degree $L = \lfloor 2N/n \rfloor + 2$, with coefficients $a^{(i)} = \mathbf{x}^{\mathrm{tr}} B^{i+1} \mathbf{z}^{\mathrm{tr}}$ in the ring $\mathsf{K}^{n\times n}$.

The next stage, called the minpoly stage, finds the minimum-degree linear recurrence $\Lambda(\lambda)$ that generates the sequence $a^{(0)}, a^{(1)}, \ldots, a^{(L)}$. In Wiedemann's original algorithm, the $a^{(i)}$ are scalars and the minimum polynomial of this sequence is obtainable by the Berlekamp/Massey (Massey, 1969) algorithm. In the present context we iteratively compute the matrix polynomial

$$F_t(\lambda) = \begin{bmatrix} \Lambda_t(\lambda) \\ \Psi_t(\lambda) \end{bmatrix}$$

for $1 \le t \le L$. The coefficients of the polynomials Λ and Ψ are $n \times n$ matrices over K. At each iteration t, the first n rows of a discrepancy matrix Δ_t, defined as the coefficient of λ^t in $F_t(\lambda)A(\lambda)$, are set to zero by elementary row operations performed in a sequence determined by an integer weight associated with each row of $F_t(\lambda)$. Then F is updated by the transformation $T_t \in \mathsf{K}^{2n\times 2n}$ along with $D = \mathrm{diag}[1, \ldots, 1, \lambda, \ldots, \lambda]$ according to the equation $F_{t+1} = D \cdot T_t \cdot F_t$.

The final stage, called the evaluation stage, involves a Horner-like scheme to evaluate a polynomial derived from Λ whose coefficients are N-dimensional vectors. This stage yields as many as m individual vectors w which satisfy the equation $Bw = 0$. With high probability, particularly when K has large cardinality, the solution is nontrivial.

The generation of $a^{(0)}, a^{(1)}, \ldots, a^{(L)}$ costs $L + 1$ black box applications plus $O(N^2)$ arithmetic operations. The minpoly stage costs $O(nN^2)$ field operations overall. The evaluation stage costs not more than $N/n + 2$ black box calls plus $O(N^2)$ arithmetic operations, for each w produced. The reader is referred to (Coppersmith, 1994, and Kaltofen, 1994) for the complexity analysis.

We have distributed the computation of $A(\lambda)$. The νth compute node is supplied $\mathbf{x}$, B, and the νth column of $\mathbf{z}$. The νth parallel subtask computes the νth column of $a^{(i)}$ for $0 \le i \le L$. The scheduler within DSC selected target hosts from a set of approximately 30 machines which had diverse processing power and memory capacity. It used estimated processor and storage requirements of the subtasks in making the selections.

3.2. The Generic-Arithmetic Feature

Our programs are all written in the C programming language to run under the UNIX operating system. Arithmetic operations are done generically through macro calls. We have created libraries of macros for arithmetic in $GF(2^k)$ and in $GF(p)$ where $1 \leq k \leq 31$ and $\log_2 p \leq 15$. The solver can link to either of these libraries at compile time and is specialized at run time when the values of k or p are supplied by the user. We have also written an optimized version specifically for GF(2).

Arithmetic operations in $GF(2^k)$ are implemented with the bit operations *exclusive-or*, *and*, and *shift* available in the C language. Field division is done by multiplying the dividend by the inverse of the divisor. The inverse of a field element is computed with the extended Euclidean algorithm within which bit operations are used for finding quotients and remainders. We use an unsigned long word to represent an element.

In GF(2), multiplication and addition are respectively the bit-operations *and* and *exclusive-or*. A single bit is sufficient to represent an element and so, thirty-two bit-vectors can be packed into a single machine word. All bit-operations are applied simultaneously across all the bits in a machine word. Due to this "double blocking" the effective blocking factor for n subtasks is $32n$ in the sequence generation stage.

In the generic field $GF(p)$ we use built-in integer arithmetic operations to implement field arithmetic. The restrictions on k and p arise from the maximum size, in bits, of a word in a machine-representation. The bound of 15 bits on $\log_2 p$ ensures that any intermediate product can safely be accommodated in an unsigned 32-bit integer. This forces the maximum value of p to 32749.

3.3. Experiments and Observations

We report on experiments with two systems of equations over GF(32749) and three systems over GF(2). The coefficient matrices in all cases are square. Case 1, over GF(32749), has dimension 10,000, contains between 23 and 47 non-zero entries per row, and a total of 350,000 such elements. It was generated by the random placement of random non-zero field elements in its rows.

Case 2, also over GF(32749), has dimension 20,000, contains 57 to 73 non-zero entries per row or about 1.3 million entries in total. It is also a "random" square matrix. This matrix and the one in the previous example are available upon request from the third author.

Case 3, over GF(2), with dimensions 52,250 × 50,001 contains 9 to 34 entries per row and 1.1 million entries in total. It arose from integer factoring by the MPQS method and was supplied to us by A. M. Odlyzko. We solved the transposed system of equations, made square by appropriate padding with rows of zeros.

Case 4, over GF(2), with dimension 100,000, contains 89 to 117 non-zero entries per row or about 10.3 million entries in total. It is a "random" square matrix.

Case 5, over GF(2), with dimensions 245,811× 252,222 containing 10 to 217 non-zero entries per row or about 11.04 million entries in total. It is the matrix used by A. K. Lenstra in the process of factoring the RSA-120 challenge number by the PPMPQS method. As in Case 3 we solved the transposed, padded system.

In the $GF(p)$ cases We used blocking factors of 2, 4, and 8. For the others, we used blocking factors of 32, 64, and 96 respectively. In each experiment, the Sequence generation stage was distributed under DSC. Vectors and the matrix were passed as files.

N	Task	Blocking Factor 2	4	8
10,000	sequence	$7^h29'$	$3^h54'$	$2^h09'$
	minpoly	$2^h25'$	$4^h08'$	$8^h00'$
	evaluation	$3^h47'$	$1^h59'$	$1^h05'$
	total time	$13^h41'$	$10^h06'$	$11^h14'$
	work	$607^{\#}$	$681^{\#}$	$751^{\#}$
20,000	sequence	$57^h17'$	$28^h43'$	$15^h21'$
	minpoly	$9^h48'$	$16^h36'$	$33^h39'$
	evaluation	$29^h42'$	$14^h44'$	$7^h53'$
	total time	$96^h47'$	$60^h02'$	$56^h53'$
	work	$4413^{\#}$	$4330^{\#}$	$6366^{\#}$

Figure 4: Parallel CPU Time (hours^{h}minutes$'$) for different blocking factors with all arithmetic in GF(32749). Each processor is rated at 28.5 MIPS. Work is measured in units of (MIPS-hours$^{\#}$).

N	Task	Blocking Factor 1×32	2×32	3×32
52,250	sequence	$3^h53'$	$2^h11'$	$1^h37'$
	minpoly	$2^h30'$	$3^h09'$	$3^h54'$
	evaluation	$1^h15'$	$0^h33'$	$0^h22'$
	total time	$7^h38'$	$5^h53'$	$5^h53'$
	work	$219^{\#}$	$231^{\#}$	$261^{\#}$
100,000	sequence	$77^h37'$	$44^h05'$	$27^h28'$
	minpoly	$10^h03'$	$12^h28'$	$15^h42'$
	evaluation	$74^h37'$	$27^h48'$	$11^h09'$
	total time	$162^h17'$	$84^h31'$	$54^h19'$
	work	$4625^{\#}$	$3741^{\#}$	$3045^{\#}$
252,222	sequence	$169^h33'$	$91^h50'$	$70^h05'$
	minpoly	$70^h29'$	$52^h50'$	$71^h25'$
	evaluation	$65^h35'$	$36^h50'$	$26^h25'$
	total time	$305^h39'$	$181^h30'$	$167^h55'$
	work	$8649^{\#}$	$7735^{\#}$	$6738^{\#}$

Figure 5: Parallel CPU Time (hours^{h}minutes$'$) for different blocking factors with all arithmetic in GF(32749). Each processor is rated at 28.5 MIPS. Work is measured in units of (MIPS-hours$^{\#}$).

Since a shared file system was in use, no physical transfer of files was necessary and only file names were exchanged. We report the parallel time with respect to a Sun-4 machine rated at 28.5 MIPS. Figures 4 and 5 below give the parallel CPU time taken for each task. The time reported for evaluation is the time to find the first non-zero solution. We also give the work performed in each experiment, which is the sum of the number of instructions performed by all processors.

In both tables the time to compute the sequence decreases with increased n. In Case 1 and Case 2 it drops approximately in half each time the blocking factor doubles. The

overall trend is still visible in the other cases but not as clearly because of the overhead associated with unpacking the doubly blocked bit-vectors.

The quadratic time complexity of the computation of the minimum polynomial is also observed. Increasing n for a given system, brings an improvement in the total time upto a point because of the shorter sequence generation and evaluation stages. The overhead of multiplying $n \times n$ matrices during the update phase can overwhelm the savings in the other stages. This stage can become a bottleneck when n is large and the black box application is fast. Conversely when the the black box application is expensive, it makes sense to use a large n. This was observed in another set of experiments involving the Block Wiedemann solver in polynomial factorization (Kaltofen & Lobo, 1994). There sequence-generation is the bottleneck.

Horner-like evaluation in the third stage costs $O(N/n)$ and we report the time necessary for finding the first non-zero solution. This stage takes approximately half as much time as the sequence-generation stage. There is no guarantee the solution w will be nontrivial or that any two nonzero solutions will be different.

Coppersmith implemented his algorithm in FORTRAN on an IBM 3090 mainframe computer and reports a time of $1^h05'$ for a system of dimension $65,518$ containing approximately 1.3 million non-zero entries. A block, lookahead Lanczos algorithm (Coppersmith, 1991) took a time of $1^h05'$. Another system of size $82,469$ containing 3.9 million entries was solved by him using the Lanczos algorithm, with some pre-processing, in $4^h52'$ on an RS 6000 workstation and in $2^h30'$ on the IBM 3090. With some reservations, we estimate that his implementation of the Blocked Wiedemann algorithm would also take $4^h52'$ on the workstation. We also solved Case 3 using the original ($n = 1$) Wiedemann algorithm in about 114 hours of CPU time.

It is important to remember that DSC ran the jobs in background-mode where they consumed machine cycles that would otherwise be wasted. Our scheduler met the minimum goal of sending one subtask at a time to a target machine. Machines with high power and memory capacity were declared to be temporarily unusable if they were under high load conditions at the time of distribution. Hosts with unused surplus capacity were identified and sometimes given more than one task to process. The distributed tasks were monitored and hosts that became inactive were identified. In an exceptional situation that we observed, the scheduler correctly determined that two machines had ceased to operate and it successfully restarted their tasks on other active machines in its list.

4. Conclusions

Using intelligently scheduled parallel subtasks in DSC we have been able to prove titanic integers prime. We have also been able to solve sparse linear systems with over 10.3 million entries over finite fields. Both tasks have been accomplished on a network of common computers. We have solved a linear system with over 245,000 equations, over 250,000 unknowns, and over 11 million non-zero entries over GF(2). The challenge we propose is to solve such systems over GF(p) for word-sized primes p, and ultimately over the rational numbers. In order to meet our challenge, we will explore several improvements to our current approach, by which we hope to overcome certain algorithmic bottlenecks in the block Wiedemann algorithm. As Figure 4 shows, higher parallelization of step (1) slows step (2). One way to speed step (2) with high blocking factor is to use a blocked Toeplitz linear system solver (Gohberg *et al.,* 1986) instead of the generalized Berlekamp/Massey algorithm. The latter method can be further improved to

carry out step (2) in $O(n^2 N(\log N)^2 \operatorname{loglog} N)$ arithmetic operations using FFT-based polynomial arithmetic and doubling (see Bitmead and Anderson, (1980); Morf, (1980), and Kaltofen, (1994)).

Another way to speed step (2) is to set up a pipeline between the subtasks that generate the components of $a^{(i)}$ and the program that computes the linear recurrence. Each subtask would compute a segment of $M \leq 2N/n$ sequence elements at a time, and pass it on to the Berlekamp-Massey program which could begin working on these terms of A. Meanwhile the subtasks would compute the next M terms of the sequence. We plan to use co-routines to implement this pipeline.

If the computation of the product of the coefficient matrix of the system by a vector is more costly, as is the case in our new application to polynomial factoring (Kaltofen and Lobo, 1994), additional measures can be taken for speed-up. For instance, the use of distinct row and column blocking factors with $m \gg n$ decreases the length L of the required matrix sequence while increasing the arithmetic overhead. Second, the number of matrix-times-vector products in the evaluation step can by reduced by storing intermediate results of the sequence step (see Kaltofen, 1994, Appendix B).

We recently redesigned our code for solving black box linear systems in the C++ programming language. In doing so, we have linked into a big integer package by A. K. Lenstra. It is also possible to link to other such packages. Thus, we now have the ability to solve sparse linear systems modulo a prime number p without any size limitation, in bits, on p.

Note added in proof: We have begun to investigate the use of the process queue length on each computer as a parameter in the scheduler.

References

Atkin, A. O. L., Morain, F. (1993). Elliptic curves and primality proving. *Math. Comput.* **61**/203, 29–68.

Batut, C., , Bernardi, D., Cohen, H., Olivier, M. (1991). User's Guide to PARI-GP. *Manual.*

Bitmead, R. R., Anderson, B. D. O. (1980). Asymtotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra Applic.* **34**, 103–116.

Char, B. W. (1990). Progress report on a system for general-purpose parallel symbolic algebraic computation. *Proc. 1990 Internat.Symp. Symbolic Algebraic Comput.*, ACM Press: S. Watanabe and M. Nagata eds., 96–103.

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

Coppersmith, D. (1991). Solving linear systems over GF(2). *Tech. Report* **RC 16997** IBM Thomas J. Watson Research Ctr., Yorktown Heights, New York.

Coppersmith, D. (1994). Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.***62**/205, 333–350.

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

Diaz, A., Kaltofen†, E. Schmitz, K., Valente, T. (1991). DSC A System for Distributed Symbolic Computation. *Proc. 1991 Internat. Symp. Symbolic Algebraic Comput.* ACM Press: S. M. Watt editor, 323-332.

Gohberg, I., Kailath, T., Koltracht, I. (1986). Efficient solution of linear systems of equations with recursive structure. *Linear Algebra Applic.* **80**, 81–113.

Hong, H., Schreiner, W. (1993). A new library for parallel algebraic computation. *Sixth SIAM Conf. Parallel Processing for Scientific Computing* **2**, Sincovec, R. F., *et al.,*editors, 776–783.

† The papers by Kaltofen (et al.) are also available from `anonymous@ftp.cs.rpi.edu` in directory `kaltofen`, or via Kaltofen's xmosaic homepage at the URL address `http://www.cs.rpi.edu/~kaltofen`, or via the gopher server at the internet address `cs.rpi.edu` (see directory `ftp/kaltofen`).

Kaltofen, E. (1995). Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comput.* to appear.

Kaltofen, E., Lobo, A. (1994). Factoring high-degree polynomials by the black box Berlekamp algorithm. *Proc.Internat. Symp. Symbolic Algebraic Comput. ISSAC '94* ACM Press: J. von zur Gathen and M. Giesbrecht editors, 90–98.

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

Valente, T., Yui, N. (1989). An improved Las Vegas primality test. JournalProc. ACM-SIGSAM 1989 Internat. Symp. Symbolic Algebraic Comput., ACM Press, 26–33

Kogge, P. M. (1991). The Architecture of Symbolic Computers. *McGraw-Hill.*

Massey, J. L. (1969). Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory* **15**, 122–127.

Morain, F. (1991). Distributed primality proving and the primality of $(2^{3539} + 1)/3$. *Advances in Cryptology—EUROCRYPT '90.* Springer Lect. Notes Comput. Sci., I. B. Damgård editor, **473**, 110–123.

Morf, M. (1980). Doubling algorithms for Toeplitz and related equations. *Proc. 1980 IEEE Internat. Conf. Acoust. Speech Signal Process.*, 954–959.

Seitz, S. (1992). Algebraic computing on a local net. *Computer Algebra and Parallelism* Springer Lect. Notes Math. **584**, 19–31.

Valente, T. (1992). A distributed approach to proving large numbers prime. *Ph.D. Thesis*, Dept. Comput. Sci., Rensselaer Polytech. Instit., Troy, New York.

Wiedemann, D. (1986). Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory* **32**, 54–62.