$\{0,1\}$-Solutions of Integer Linear Equation Systems*

Anton Betten, Alfred Wassermann

Department of Mathematics
University of Bayreuth
Germany

Abstract. A parallel version of an algorithm for solving systems of integer linear equations with $\{0,1\}$-variables is presented. The algorithm is based on lattice basis reduction in combination with explicit enumeration.

1 The algorithm

A parallel version of an algorithm proposed by Kaib and Ritter [4] has been implemented with PVM to find all $\{0,1\}$-solutions of integer linear equation systems. For example such systems are of interest in the construction of block designs, see [1, 2, 3, 8]: It is possible to find block designs if one finds $\{0,1\}$-vectors x and $\lambda > 0$ with

$$A \cdot x = \lambda(1,1,\ldots,1)^{\top}, \tag{1}$$

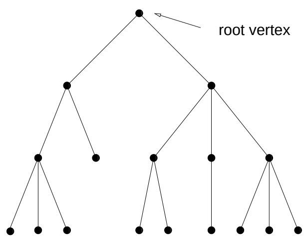
where A is a matrix consisting of nonnegative integers. Our problem is also related to cryptography [6] and theory of numbers [7].

The algorithm – using lattice basis reduction [5] – constructs a basis for the equation kernel that consists of short integer vectors. Then the integer linear combinations of these basis vectors are enumerated and tested if they yield $\{0,1\}$-solutions of (1). For an explicit description of the algorithm see [8].

2 Parallelization

The backtracking algorithm is now implemented using PVM. The algorithm runs along a search tree with an unpredictable number of childs at each vertex. Actually, we search all solutions of a discrete optimization problem, the optimal value of the objective function being well known in advance.

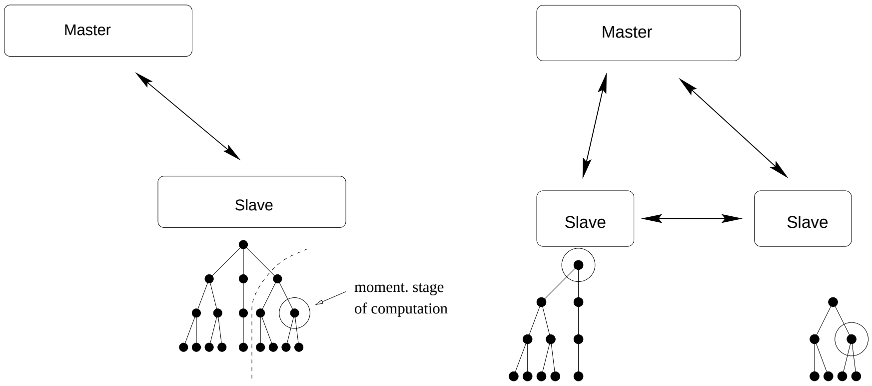
* This research was supported by Procope.



The parallel version of the backtracking algorithm is easily given: We fix the maximal number of tasks allowed to be processed by the virtual machine. This number depends for instance on the number of machines or on the amount of main memory available.

There is a principal task controlling every other task, that we will call "master". After some preprocessing, i.e. LLL reduction, the master creates subordinate tasks, called "slaves", that possess their own search loop. Each slave enumerates a certain part of the search tree. After enumerating all branches of its subtree the slave is allowed to die.

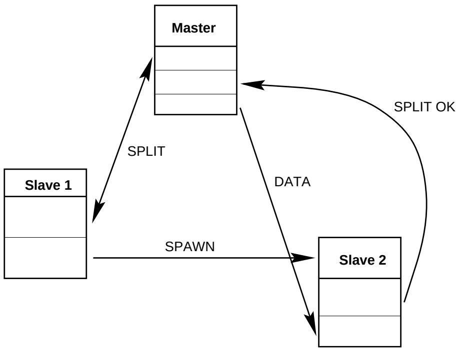
If the fixed maximal number of tasks is not reached during some stage of the algorithm, a new task will be created by the master. Therefore, the slave who seems to have gained the least progress is told to *split*, i.e. to create a new slave, who does part of the work of the old task. The search tree of the old task is shrunken.



The slave who got the split message still finishes the branch of the root tree in which he is currently computing (rightmost branch in the figure). But just before doing so he creates a new slave and initiates a task containg the remaining branches (of the root vertex). So, the original tree of the slave is divided into two parts, the momentary subtree is cut off and will be finished by the slave himself, the remaining branches are carried over to the new slave. Note that splitting is always done at the level of the root vertex. Otherwise, the shape of the subtrees would become too difficult to handle. Here, the root vertex and its first edge to a deeper node is all one needs to know for defining a subtree: the tree is defined to be all those subtrees of the root vertex which start at the special edge and continue with the following branches – remember that we enumerate a basis of a kernel so there is a notion of left and right in each level. The new slave informs the master of his existence and both slaves start to work on their two smaller subproblems.

Implicitely, this strategy implies dynamical load balancing: Each machine receives a new task as soon as it has finished the previous one. This reduces the size of other trees. Therefore slower machines get help by faster machines.

Note that perhaps a split request cannot be carried out. Namely, if the search (sub-) tree does not possess any more branches leading down from the root vertex except that one containing the momentary position of computation. The next figure shows the messages needed for a task split.



In a first version of our algorithm, we made PVM to choose by itself the machine where to spawn the new task. But we noticed bad behaviour of the algorithm, because some machines got too much load. Therefore we decided to put machines and tasks under control of the master. So each split message of the master names a certain machine where the new task should be spawned.

3 Results

Since we do not have an homogenous pool of computers we indicate the speed of each computer of our virtual machine by percentage of the speed of a Pentium 90 running under Linux. In order to measure the running time, each computer of the virtual machine was tested with the serial version of the program, computing just a small example.

machine type	P90 speed
HP 9000 755 / 99 MHz	303 %
HP 9000 712 / 80 MHz	240 %
Silicon Graphics SGI5	181 %
Intel Pentium 90 MHz	100 %

Several tests with PVM were done with the input matrix `KM_PGL23plus_t6_k8` and $\lambda = 36$ from [2]. This is a 28×119 matrix with dimension of the search space equal to 93. Each solution is a 6-(25,8,36) design. For a detailed explanation of the automorphism group see also [2]. The following table lists the results of our tests on four different configurations of virtual machines. The second column lists the number of computers of each type we used. The third column contains the types of these computers. The fourth column contains the collected percentage of Pentium 90 speed of the computers in the virtual machine. In the column "No. Proc." the maximal number of slave processes is noted which were allowed to run simultaneously on the virtual machine. The last column gives the time after which the result was printed on the screen.

	No.	Computer	P 90 speed	No. Proc.	Time
1.	3	Silicon Graphics SGI5	181 %		
	3		543 %	12	323 min
2.	2	HPPA 9000/755 99 MHz	303 %		
	1	HPPA 9000/712 80 MHz	240 %		
	2	Silicon Graphics SGI5	181 %		
	5		1208 %	24	160 min
3.	3	HPPA 9000/755 99 MHz	303 %		
	1	HPPA 9000/712 80 MHz	240 %		
	6	Silicon Graphics SGI5	181 %		
	10		2235 %	45	89 min
4.	6	HPPA 9000/755 99 MHz	303 %		
	1	HPPA 9000/712 80 MHz	240 %		
	6	Silicon Graphics SGI5	181 %		
	13		3144 %	58	56 min

Moreover, with the fourth configuration we were able to find 10008 solutions for the same matrix `KM_PGL23plus_t6_k8` and $\lambda = 45$ which were previously not known to exist. The computing time we needed was 7:34 hours.

References

1. A. Betten, A. Kerber, A. Kohnert, R. Laue, A. Wassermann: The Discovery of Simple 7-Designs with Automorphism Group $P\Gamma L(2, 32)$. *AAECC 11* in *Lecture Notes in Computer Science* **547** (1995), 281–293.
2. A. Betten, R. Laue, A. Wassermann: Simple 7-Designs With Small Parameters, Spetses, 1996.
3. D. L. Kreher, S. P. Radziszowski: Finding Simple t-Designs by Using Basis Reduction. *Congressus Numerantium* **55** (1986), 235–244.
4. M. Kaib, H. Ritter: Block Reduction for Arbitrary Norms. Preprint 1995.
5. A. K. Lenstra, H. W. Lenstra Jr., L. Lovász: Factoring Polynomials with Rational Coefficients, *Math. Ann.* **261** (1982), 515–534.
6. J. C. Lagarias, A. M. Odlyzko: Solving low-density subset sum problems. *J. Assoc. Comp. Mach.* **32** (1985), 229–246.
7. C. P. Schnorr: Factoring Integers and Computing Discrete Logarithms via Diophantine Approximation. *Advances in Cryptology – Eurocrypt '91* in *Lecture Notes in Computer Science* **547** (1991), 281–293.
8. A. Wassermann: Finding Simple t-Designs with Enumeration Techniques, submitted.

This article was processed using the LaTeX macro package with LLNCS style