Parallel Computation in Combinatorics using PVM

Anton Betten
Universität Bayreuth

Sébastien Veigneau
Université de Marne-la-Vallée

Alfred Wassermann
Universität Bayreuth

Parallel Computation in Combinatorics[1] using PVM

Anton Betten
Universität Bayreuth

Sébastien Veigneau
Université de Marne-la-Vallée

Alfred Wassermann
Universität Bayreuth

Abstract. We give several examples where PVM was successfully used as a tool for distributed computation of solutions of combinatorial problems. Involved topics are the computation of the solution of multi-dimensional subset sum problems which appear in the construction of block designs, the construction and classification of finite solvable groups up to isomorphism, and the computation of spin polynomials that are generating functions of ribbon tableaux and which lead to generalizations of Hall-Littlewood functions.

Résumé. Nous donnons plusieurs exemples pour lesquels nous avons utilisé PVM comme outil de distribution de résolution de problèmes combinatoires. Les sujets abordés sont la résolution du problème du sac-à-dos multi-dimensionnel qui apparaît dans la construction des "*block designs*", la construction et la classification groupes finis résolubles à isomorphismes près, et le calcul des polynômes de spin qui sont des fonctions génératrices de tableaux de rubans qui conduisent à des généralitions des fonctions de Hall-Littlewood.

Zusammenfassung. Mehrere Beipiele für den Einsatz von PVM als Hilfsmittel bei der verteilten Berechnung von Lösungen kombinatorischer Probleme werden diskutiert. Themen sind die Berechnung von Lösungen mehrdimensionaler Rucksackprobleme, welche bei der Konstruktion von Blockplänen auftreten, die Konstruktion und Klassifikation bis auf Isomorphie von endlichen auflösbaren Gruppen, sowie die Berechnung von Erzeugendenfunktionen für Randhaken-Tableaux bei der Verallgemeinerung von Hall-Littlewood Funktionen.

(1) This research was supported by Procope.

Table of contents

Abstract, Résumé, Zusammenfassung **i**

1 Introduction **1**

Parallel Computing 2

PVM 4

2 PVM programs **7**

$\{0, 1\}$-Solutions of Integer Linear Equation Systems 7

t-Designs 7

Lattice Basis Reduction 9

Explicit Enumeration 9

Parallelization 13

Results 15

Construction of Finite Solvable Groups 18

Group Extensions 18

Reductions 22

Parallelization 23

Results: Serial vs. Parallel Version 25

Ribbon tableaux and spin polynomials 28

Introduction 28

Spin Polynomial 30

Matrix Coding of Ribbon Tableaux 31

Recursive Computation of Spin Polynomials 32

Parallel implementation 35

Results 36

References **43**

1 Introduction

In the past, scientific computing meant large sequential calculations typically done on single huge machines. As computers are getting more and more connected via world-wide networks a new chance of gaining computing power from the net arises: a collection of lots of even small computers can assemble computing power which exceeds that of the largest machines in the world. Other approaches would be shared-memory multiprocessors, systolic arrays or massive SIMD machines. It is clear that the full potential of such systems cannot be realized without similar advances in parallel software. The software has to be adapted to the new situation: New versions of the algorithms have to be developed, existing algorithms have to be modified and completely new algorithms have to be invented. The programs have to take into account the distribution of the computation onto different machines. In fact, the multiprocessor approach introduces three new requirements not encountered before:

- Each problem must be partitionned into tasks;
- Each task must be scheduled for execution on one or more processors;
- Synchronization of control and data flow must be performed during execution.

Looking to the future, some applications will be good candidates for parallel computing. However, to run on a parallel machine or to write code for a parallel machine, you need to think about how the program can be split up into pieces. At the moment, there is no general way to parallelize an algorithm efficiently. Each algorithm has to be analysed independently in order to find the most appropriate model of parallelization. For some applications, a parallel approach is natural, for others the benefit will be very little.

The field of combinatorics gives rise to many applications where a parallel approach is very fruitful. In this paper we describe, for three different algorithms in combinatorics, three different ways of parallelization in a distributed computing model. More explicitly, we want to enumerate combinatorial structures. The aims were:

1. To compute $\{0,1\}$-solutions of integer linear equation systems which arise for example at the construction of block designs.
2. The construction and classification of discrete structures such as finite solvable groups up to isomorphism.
3. The evaluation of spin polynomials, counting ribbon tableaux recursively.

These problems were considered in a joint effort at the universities of Bayreuth and Marne-la-Vallée. We used the package PVM which is available for free on the internet. PVM allows to combine hardware of different manufacturers and is well suited for an installation on existing computer pools. We used it on pools at Bayreuth and Marne-la-Vallée and also experimented with a combination of the machines of both sites.

Parallel Computing

In order to achieve more speed in computation the traditional model of computers basically invented by von Neumann shows its limitations more and more. Progress of technology was always very fast in hardware and concerns for instance techniques of storage in memory and on hard discs. Primarily, also the capabilities of the computers central processing unit, more commonly known as processors or CPU (central processor unit), are the target of much innovations. One can say that the speed of processors was almost doubled in each five years in the past three decades. These achievements were possible by integrating more and more electronical circuits onto the surface of processors which is burned onto silicium wafers of only few square millimeters size. But there are limitations due to the physical nature of the material. Obviously one cannot decrease the size of the electronic circuit up to an infinitely thin connection. There are other technologies such as optical circuits or quantum computers but one cannot estimate if they will be available in reasonable time and with acceptable prices.

In recent times, the development was led more and more in the direction of parallel computation. Here one tries to avoid the bottleneck of the central processing unit and lets many processors take part in the computation. But how can such a model be realized ? As of now, there are two basic architectures, called SIMD (single instruction, multiple data) and MIMD (multiple instructions, multiple data). We will roughly sketch the first in a few sentences, the other one will be treated throughout the whole article and examples of different programs will be discussed.

The technology of so called “massively parallel computers” (for example MasPar or Connection Machine of Thinking Machines) provides access to a large amount of processors arranged in a specific way, for instance as an array of processors (processor array or PE on MasPar) or as hypercube (connected are exactly those processors which have their Hamming distance equal to one). Processors can communicate with each of their neighbours very fast. There is also the possibility to exchange data globally, i.e. for example with the central unit or front-end computer which is included also. Global communication between the parallel processors is possible but limited in the number of communications and also in speed. As was already said, local communication is always very fast compared to global communication. But the architecture has another drawback which is indicated already by the name: there is always one single instruction which is executed simultaneously on the processors. One has the notion of the active set of processors, that is the set of those processors which take part in the execution of the specific instruction which will be next. The parallel concept is realized by letting each processor work on its own set of data. Therefore the variables are the same on all processors but each processor has a specific instance held in its local memory, also known as the processor memory. The machines can be programmed using a high level language, for example on MasPar there is a C-language which extends usual capabilities by the introduction of plural variables and plural statements. Plural variables exist on each processor (with each processor having its specific values in it). Plural statements execute only on the processors in the active set.

For the old-fashioned programmer of serial machines it might be interesting to see that for example an `if-then-else` statement is executed in both of its branches. Only those processors which evaluate the controlling expressing to true execute the first (`if`) branch. The complementary set of processors execute the `else` part of the statement. Here the complement is taken in the set of active processors as it was at the time just before the execution of the `if` statement. There are plural `while` and `for` loops also. Such loops are iterated as long as there is at least one processor left *i.e.* which evaluates the controlling expression to true. Note carefully that it is perfectly possible to have the loop variable as a plural variable. So each processor can have different ranges of the loop iterations.

After this short overview of the architecture of SIMD machines, one might guess that there are interesting possibilities in programming such computers. But one has to admit also that too much parallelism can lead to difficult programming tasks. Such programs can be difficult to write and difficult to debug. One word on automatic writing of parallel programs has to be included. There are approaches to implement parallel compilers *i.e.* compilers which compile programs for a parallel machine. These compilers are supposed to compile usual high level language code such as C or Modula source for example. They try to detect loops in the program which can be parallelized and do the adaptation to the parallel machine automatically. But one can imagine that in the general case it can be a very difficult task to analyze sequential programs for possible parallelism. Sometimes only the author of the program (or the author of the algorithm for the problem) knows exactly what to parallelize *i.e.* how to distribute parts of the problem for parallel execution. In any of these cases one has to program the parallel machine by hand.

In contrast to the SIMD concept where the single processors are almost very weakly equipped computers in MIMD we have quite a different situation. Now the single elements are fully equipped workstations which typically have large main memory and local disks. They are connected by a network and they can run independently from each other. Each workstation can have several tasks running and the network is able to connect computers which are far away from each other. Communication is needed to exchange data and for synchronization between processes. The messages are like electronic mail between processes and can contain data or synchronization commands. Messages can be sent asynchronously or blocking, i.e. the receiving process waits for a particular message to arrive. One has to be careful whether to use blocking or non-blocking receive. The first gives control back to the deamon so that the task is temporarily sleeping. The other method should be used if one has other things to do in the task (for example a computation). One uses non-blocking receive for example just like looking into ones mailbox to see if new mail has arrived. But in most cases one does not wait and stay there until the next mail comes in. For computers, blocking receive has the advantage that control goes back to the daemon. Thus the task uses no computing time during sleeping.

In most cases, one builds message loops into the programs so that they can react appropriately to each message. Thus one tries to avoid states in the program. This is the same like programming a X-Windows event queue where one has to react for the commands of the user interacting via menus or windows with the program.

The collection of computers build a big "virtual machine" and this is where PVM (parallel virtual machine) has its name from. In the next section we continue with a short introduction on PVM.

PVM

PVM is a library of C or FORTRAN callable functions devoted to distributed computing. It enables a collection of heterogeneous computers to be viewed as a single **P**arallel **V**irtual **M**achine. The programming interface allows the initiation and termination of tasks and provides routines to handle communication (synchronization) between tasks. PVM transparently handles all message routing, data conversion, and task scheduling across a network of possibly incompatible computer architectures. PVM uses existing networks (Ethernet, FDDI, ATM, ...) and does not impose a special structure (ring, star, tree, grid, ...). PVM runs on most Unix platforms (DEC alpha, HP, IBM, Linux, NeXt, SGI, SUN, ...), on several massive parallel systems like CM2, CM5, MasPar, on vector computers (Cray, Convex) and there are ports to OS/2 and MS-Windows. Programs compiled on all these different architectures can communicate which each other. The communication of the tasks is managed by PVM demons running on each computer of the virtual machine. The user decides which machines are incorporated into the virtual machine that is determined by an ASCII file. This file contains information about each component of the virtual machine; each physical computer has to be included.

The atomic structure of a PVM program is a *task* (normally one Unix process): an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied by PVM; multiple tasks may execute on a single processor.

The hardware environment may be viewed by the user as an attributeless collection of virtual processing elements or may be divided into a set of different working groups where each group executes the most appropriate tasks for its capabilities.

The programmer divides his algorithm into several tasks and explicitly administrates these tasks, *i.e.* he initiates, terminates and handles the communication between the tasks. Each task can communicate with each other task. Also broadcasting to all tasks or a group of tasks is possible.

Each task is identified by a specific number, called task id, which is unique in the virtual machine. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

Example: The following small example executes the most frequently used PVM routines. It consists of two programs: a master and a slave task. The master program starts the slave task which just sends the string "I'm a slave" to the master and then exits. The master then prints the received message to the screen.

`pvm_mytid()` returns the task id of the program itself. After printing its own task id the program initiates another task: the program `task` is started with the routine `pvm_spawn()`. This routine has several parameters which give the user control where the task is started. In `tid` the task id of the new task is returned.

After starting the new program, the main program waits for a message of this new task. The routine `pvm_recv(tid,msgtag)` waits until a message of type `msgtag` from the task with task id `tid` arrives. After arrival of the message, `pvm_upkstr()` writes a string, which is expected to be sent by the subtask, into the local string `buf`. Finally `pvm_exit()` disconnects the program from the virtual machine.

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
void main(void)
{
  int cc, tid, msgtag, mtid;
  char buf[100];

  /* Declaring myself as a task in the PVM */
  mtid = pvm_mytid();
  printf("Main task (Master). task id:%x\n", mtid);

  cc = pvm_spawn("task",(char**)0,0,"",1,&tid);
  if (cc==1) {
    msgtag = 1;
    pvm_recv(tid,msgtag);
    pvm_upkstr(buf);
    printf("Message from task id %x: %s\n",tid, buf);
  } else {
    printf("Something's wrong");
  }

  pvm_exit();
}
```

The following program `task` is started by the main application. `pvm_parent()` returns the task id of the task which started the current program. `pvm_initsend()` prepares the buffer which will contain the information we want to transfer. The function `pvm_pkstr()` writes a string into this buffer and `pvm_send(ptid,msgtag)` sends the buffer to the task with task id `ptid` and labels the message with `msgtag` (Note that the main task is waiting for message of type `msgtag=1`). Finally the subtask is removed from the virtual machine with `pvm_exit()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
void main(void)
{
  int ptid, msgtag, mtid;
  char buf[100];

  mtid = pvm_mytid();
  ptid = pvm_parent();

  msgtag = 1;
  pvm_initsend(PvmDataDefault);
  strcpy(buf,"I'm a slave");
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

The programs are compiled on each computer included in the virtual machine during the execution of our program. The following lines give an example of the execution of the master program:

```
$ master
Main task (Master). task id:40001
Message from task id 40002: I'm a slave
```

2

PVM programs

This chapter clusters three PVM programs for which parallel approach has been fruitful. The first section details the computation of the $\{0,1\}$-solutions of integer linear equation systems, the second one explains how to construct finite solvable groups and the last one gives an algorithm to recursively enumerate ribbon tableaux.

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

t-Designs

Let X be a v-set (i.e. a set with v elements) whose elements are called *points*. A t-(v,k,λ) *design* is a collection of k-subsets (called *blocks*) of X with the property that any t-subset of X is contained in exactly λ blocks. A t-(v,k,λ) design is called *simple* if no blocks are repeated, and *trivial* if every k-subset of X is a block and occurs the same number of times in the design.

A straightforward approach to the construction of t-(v,k,λ) designs is to consider the matrix

$$M^v_{t,k} := (m_{i,j}), \quad i = 1,\ldots,\binom{v}{t},\ j = 1,\ldots,\binom{v}{k}:$$

The rows of $M^v_{t,k}$ are indexed by the t-subsets of X and the columns by the k-subsets of X. We set $m_{i,j} := 1$ if the i-th t-subset is contained in the j-th k-subset, otherwise $m_{i,j} := 0$. Simple t-(v,k,λ) designs now correspond to $\{0,1\}$-solutions x of the system of $\binom{v}{t}$ linear equations:

$$M^v_{t,k} \cdot x = \lambda(1,1,\ldots,1)^\top.$$

Unfortunately, for most designs with interesting parameters v, t, k the size of the matrix $M^v_{t,k}$ is prohibitively large. For example in the case of $v = 33$, $t = 7$ and $k = 8$ the matrix $M^{33}_{7,8}$ has 4 272 048 rows and 13 884 156 columns.

But by assuming a group action on the set X the size of $M^v_{t,k}$ can be dramatically reduced. A group G acting on X induces also an action on the set of t-subsets and the set of k-subsets of X. With $A_{t,k} = (a_{i,j})$ we denote the matrix where a_{ij} counts the number of those elements in the j-th orbit of G on the k-subsets of X which contain a representative of the i-th orbit of t-subsets of X. This matrix was introduced by Kramer and Mesner [8]. They observed:

Theorem 1 *(see [8]) A simple t-(v, k, λ) design exists with $G \leq \mathrm{Sym}(X)$ as an automorphism group if and only if there is a $\{0, 1\}$-solution x to the matrix equation*

$$A_{t,k} \cdot x = \lambda(1, 1, \dots, 1)^\top. \tag{2.1}$$

Taking the group $\mathrm{P\Gamma L}(2, 2^5)$ the matrix $A_{7,8}$ in the above example has 32 rows and 97 columns. Nevertheless it is still a respectable task to find <u>all</u> $\{0, 1\}$-solutions of (2.1).

Solving equation (2.1) is a special instance of the multi-dimensional *subset sum* problem which is known to be NP-complete [6]. This problem is also of interest in other areas such as cryptography [4, 11], number theory [14] and combinatorial optimization [13].

In [9] the authors used the original lattice basis reduction algorithm (LLL) as described in [12] and a lattice like the one proposed in [11]. Meanwhile lattice basis reduction algorithms have been improved and new algorithms were invented by Schnorr, [15, 16, 17]. Also new lattices have been proposed, see [5, 7].

The approach described here – using lattice basis reduction [12] – is to construct a basis of the kernel of the equation

$$\begin{pmatrix} & & 1 \\ & A_{t,k} & \vdots \\ & & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \quad x_i \in \mathbb{Z}, y \in \mathbb{Z} \tag{2.2}$$

which consists of short integer vectors. But the shortest integer vectors (in the euclidean norm) in the kernel of (2.2) need not be solutions of our $\{0, 1\}$-problem (2.1). Kaib, Ritter proposed in [10] an algorithm which enumerates all solutions with $y = \pm\lambda$ as linear combination of this short integer basis vectors.

Lattice Basis Reduction

As described in [2] and [18] we transform the Kramer Mesner matrix A_{tk} with l rows and s columns into the following matrix (2.3), where $c_1 := \lambda$.

$$\left(\begin{array}{ccc|cc} & & & c_0 1 & 0 \\ & c_0 A_{tk} & & \vdots & \vdots \\ & & & c_0 1 & 0 \\ \hline c_1 2 & & 0 & 0 & c_1 1 \\ & \ddots & & \vdots & \vdots \\ 0 & & c_1 2 & 0 & c_1 1 \\ \hline 0 & \dots & 0 & 1 & 0 \\ 0 & \dots & 0 & 0 & c_1 1 \end{array}\right) \tag{2.3}$$

We only allow integer linear combinations of the columns of the matrix (2.3). The set of all integer linear combinations of these columns, which are linear independent, is called a lattice. For an introduction to lattices see [12].

In the first step of the algorithm we use lattice basis reduction, see [12, 15, 16, 17], to compute an integer basis of the kernel of the matrix $A_{t,k}$. If c_0 is large enough, short basis vectors as constructed by these lattice basis reduction algorithms have to contain only zeros in the first l rows such that the large constant c_0 vanishes.

In the second step integer linear combinations of the basis vectors of the kernel are explicitly enumerated to find all $\{0,1\}$-solutions of the equation (2.1).

For lattice basis reduction polynomial time algorithm do exist, but the second step still needs exponential time and is now parallelized with PVM.

Explicit Enumeration

Definition 1 *Let $L \subset \mathbb{R}^m$ be a lattice of rank n. For $1 \leq p < \infty$ the norm defined by the mapping*

$$\|.\|_p : \mathbb{R}^m \to \mathbb{R},\ x \mapsto \|x\|_p := (\sum_{i=1}^{m} |x_i|^p)^{1/p}$$

is called p-norm. The norm defined by the mapping

$$\|.\|_\infty : \mathbb{R}^m \to \mathbb{R},\ x \mapsto \|x\|_\infty := \max\{|x_i| \mid 1 \leq i \leq m\}$$

is called ∞-norm.

*For $1 \leq p \leq \infty$ we call a vector $\in L$ p-*shortest *if it is a shortest nonzero vector in L in p-norm.*

Let $\langle .,. \rangle$ denote the ordinary inner product in $\mathbb{R}^n$, $n \in \mathbb{N}$. For a sequence of linear independent vectors $b_1, \dots, b_m \in \mathbb{R}^n$ we let $b_1^*, \dots, b_m^*$ be the *Gram-Schmidt* *orthogonalized* sequence. We thus have

$$b_i^* := b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \quad \text{for } i = 1, \ldots, m, \text{ where } \mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \ . \tag{2.4}$$

We set $u_{j,j} = 1$ for $j = 1, \ldots, m$.

Definition 2 *For an (ordered) basis $b_1, b_2, \ldots, b_m$ of a lattice $L \subset \mathbb{R}^n$ and $1 \le i \le m$, $\pi_i(v)$ is the* orthogonal projection *of $v \in \mathbb{R}^n$ into $\langle b_1, b_2, \ldots, b_i \rangle^\perp$.*
$L_i := \pi_i(L)$ is the orthogonal projection of the lattice L into $\langle b_1, b_2, \ldots, b_i \rangle^\perp$.

Some calculation gives

$$\pi_j(\sum_{t=j}^m u_t b_t) = \sum_{s=j}^m b_s^* \sum_{i=s}^m u_i \mu_{i,s}.$$

With $c_s := \|b_s^*\|_2^2$ for $1 \le s \le m$, it follows

$$\|\pi_j(\sum_{t=j}^m u_t b_t)\|_2^2 = \sum_{s=j}^m (\sum_{i=s}^m u_i \mu_{i,s})^2 c_s.$$

For the efficiency of the enumeration algorithm the following relation between the projections $\pi_j(\sum_{t=j}^m u_t b_t)$ and $\pi_{j+1}(\sum_{t=j+1}^m u_t b_t)$ is crucial:

$$\|\pi_j(\sum_{t=j}^m u_t b_t)\|_2^2 = (\sum_{i=j}^m u_i \mu_{i,j})^2 c_j + \|\pi_{j+1}(\sum_{t=j+1}^m u_t b_t)\|_2^2.$$

It means we only have to compute $(\sum_{i=j}^m u_i \mu_{i,j})^2 c_j \ge 0$ to get from stage $j+1$ to stage j of the algorithm.

Definition 3 *For $u_j, u_{j+1}, \ldots, u_m \in \mathbb{Z}$ we write $w_j := \pi_j(\sum_{t=j}^m u_t b_t)$.*

The backtracking algorithm tries all possible integer values for $u_m, u_{m-1}, \ldots, u_1$. Starting from $t = m$ it computes w_t for $t = m, \ldots, 1$ and finally $\sum_{i=1}^m u_i b_i = w_1$.

Remark 1 *If $u_{j+1}, u_{j+2}, \ldots, u_m \in \mathbb{Z}$ are fixed and $u_j \in \mathbb{Z}$ has to be choosen such that $\|w_j\|_2^2$ is minimal, then u_j has to be set to the nearest integer to $-\sum_{i=j+1}^m u_i \mu_{i,j}$ since*

$$\|w_j\|_2^2 = \|\pi_j(\sum_{t=j}^m u_t b_t)\|_2^2 = (u_j + \sum_{i=j+1}^m u_i \mu_{i,j})^2 c_j + \|\pi_{j+1}(\sum_{t=j+1}^m u_t b_t)\|_2^2.$$

The solutions of our system of linear equations (2.2) are the ∞-shortest vectors with $\pm c_1$ as entry in line $s + l + 1$ in the lattice generated by the columns of (2.3).

Let F be an upper bound of the p-shortest vector of L. Since all p-norms in $\mathbb{R}^n$ are equivalent, there exist constants r_p, R_p such that $r_p\|x\|_p \le \|x\|_2 \le R_p\|x\|_p$ for all $x \in \mathbb{R}^n$, for example $R_\infty = \sqrt{n}$. Therefore a p-shortest vector v has 2-norm $\|v\|_2 \le R_p F$ and in order to find p-shortest vectors we enumerate all vectors with 2-norm not greater than $R_p F$.

Moreover, Kaib, Ritter [10] use Hölder's inequality to combine the search for p-shortest vectors with enumeration in 2-norm:

Theorem 2 *If for fixed* $u_j, u_{j+1}, \ldots, u_m \in \mathbb{Z}$ *there exist* $u_1, u_2, \ldots, u_{j-1} \in \mathbb{Z}$ *with* $\| \sum_{i=1}^m w_i \|_p \leq F$ *then for all* $y_j, y_{j+1}, \ldots, y_m \in \mathbb{R}$*:*

$$\Big| \sum_{i=j}^{m} y_i \|w_i\|_2^2 \Big| \leq F \cdot \| \sum_{i=j}^{m} y_i w_i \|_q \tag{2.5}$$

with $1 \leq q \leq \infty$ *such that* $1/p + 1/q = 1$.

Proof: See Kaib, Ritter [10].

It remains to select $y_j, \ldots, y_m$ appropriately to enable an early recognition of enumeration branches which cannot yield solutions. Kaib, Ritter [10] proposed two selections:

1. $(y_j, y_{j+1}, \ldots, y_m) = (1, 0, \ldots, 0)$: Test if
$$\|w_j\|_2^2 \leq F \|w_j\|_q.$$

2. $(y_j, y_{j+1}, \ldots, y_m) = (\eta, 1-\eta, 0, \ldots, 0)$ with $\eta \in \,]0,1[$.
Let's say $w_j = x b_j^* + w_{j+1}$ for an $x \in \mathbb{R}$. Then for every successive w_j' in the same direction, that means every $w_j' = (x+r) b_j^* + w_{j+1}$ with $r \in \mathbb{Z}$ and having the same sign as x, we have for $\eta := \frac{x}{x+r}$:
$$w_j = \eta w_j' + (1-\eta) w_{j+1} \quad \text{and} \quad 0 < \eta < 1. \tag{2.6}$$
If w_j' can lead to a solution, then from (2.5) it follows for every $\eta \in \,]0,1[$:
$$\eta \|w_j'\|_2^2 + (1-\eta) \|w_{j+1}\|_2^2 \leq F \|\eta w_j' + (1-\eta) w_{j+1}\|_q. \tag{2.7}$$
With (2.6) the inequality reduces to
$$\|w_j\|_2^2 \leq F \|w_j\|_q.$$
Here $0 \leq \eta \leq 1$ is needed.
Therefore we can cut the enumeration in the direction of x if $\|w_j\|_2^2 > F \|w_j\|_q$.

This results in the following algorithm:

Algorithm 1 *1. Compute a LLL-reduced integer basis of the kernel of the linear system (2.1): Choose c_0 large enough such that the number of columns corresponding to kernel vectors will be equal to $s - l + 2$ and LLL-reduce the matrix (2.3).*

2. Remove the columns with nonzero entries in the first l rows. ¿From the remaining columns remove the first l rows (the zero entries).

3. Compute for the remaining columns $b_1, \ldots, b_m$ the Gram-Schmidt orthogonalized vectors $b_1^*, b_2^*, \ldots, b_m^*$ with their Gram-Schmidt coefficients $\mu_{i,j}$, see (2.4).

4. *Set* $j := 1$*;*
   Set $F := c_1 + \varepsilon =$ *upper limit to the* ∞*-shortest vector in* L*.*
   Set $\bar{F} := (s+2) \cdot c_1^2 = R_\infty^2 \cdot F^2$*.*

5. *Search loop:*

```
while j ≤ m
        Compute w_j from w_{j+1}.
        if ||w_j||_2^2 > F̄ then
                j := j + 1
                NEXT(u_j)
        else
            if j > 1 then
                    if PRUNE(u_j) then
                                if onedirection then
                                        j := j + 1
                                        NEXT(u_j)
                                else
                                        onedirection := true
                                        NEXT(u_j)
                                end if
                    else
                                j := j − 1
                                y := Σ_{i=j+1}^{m} u_i μ_{i,j}
                                u_j := round(−y)
                                onedirection := false
                    end if
            else /* (j = 1) */
                    PRINT u_1, ..., u_m
                    NEXT(u_j)
            end if
        end if
end while
```

The procedure NEXT determines the next value of the variable u_j. Initially u_j is set to the nearest value of $-y_j := -\sum_{i=j+1}^{m} u_i \mu_{i,j}$, say u_j^1. The next value (u_j^2) of u_j is the second nearest integer to $-y_j$ then follows u_j^3 and so forth. Therefore the values of u_j alternate around $-y_j$. If PRUNE is true for one value of u_j we do one more jump around $-y_j$ then the enumeration is only proceeded in this remaining direction until it is pruned again.

For arbitrary p with and q such that $1/p + 1/q = 1$ the procedure PRUNE looks like this:

Algorithm 2 *Choose* $y_j, \dots, y_m$

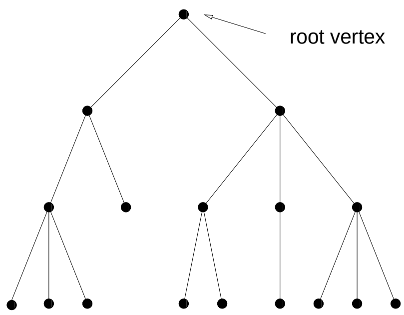
PRUNE(u_j *)*

if $\left|\sum_{i=j}^{m} y_i \|w_i\|_2^2\right| \le F \cdot \|\sum_{i=j}^{m} y_i w_i\|_q$
Return false
else
Return true
end if

Some improvements can be made in order to speed up the algorithm, see [10] and [18].

Parallelization

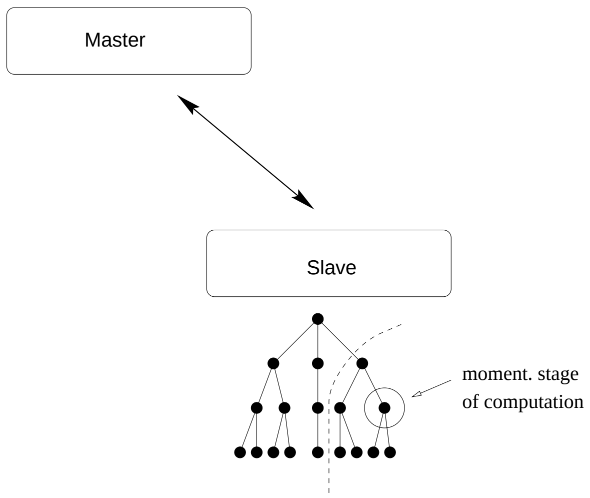
This backtracking algorithm is now implemented with PVM. The algorithm has to run through a search tree with an unpredictable number of childs in each vertex.



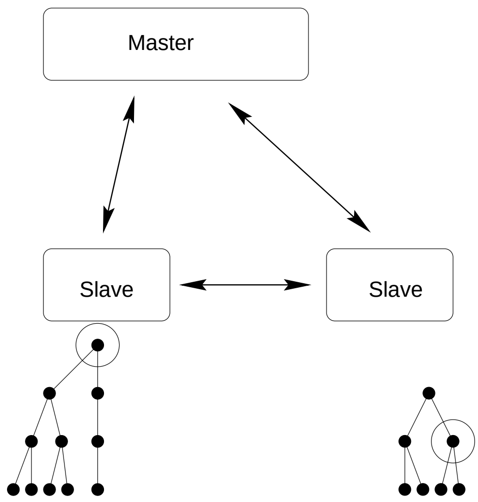
The parallel version of the above Algorithm 1 is easy to describe: We supply a maximal number of tasks which are allowed on the virtual machine. It depends on the number of machines and the amount of RAM which is available.

There is one master task which keeps control of all other tasks. After some preprocessing (step 1–4 in Algorithm 1) the master creates one slave which enters the search loop (step 5 in Algorithm 1). Each slave has to enumerate a part of the whole search tree. After enumerating all branches of the root vertex of its subtree the slave is allowed to exit.

Whenever the maximal number of tasks is not reached the master tells the slave with the least progress to *split*, i.e. to create a new task which does part of the work of the old task.



This means, the task which got the split message still has to finish the momentarily branch from the root vertex in which it is computing, but first he creates another slave which has to compute all other branches of the root vertex which were not yet computed by the original task itself.

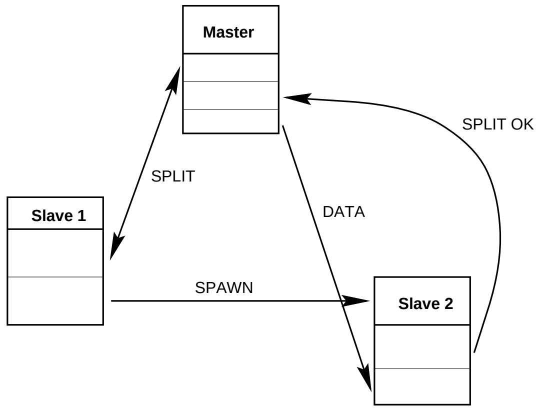


Both tasks notify the master about the creation and start to work on two smaller subproblems. This strategy implicitely implies dynamical load balancing. Each machine gets a new task as soon as it has finished the previous one. Therefore slow machines get help by the fast machines. At the moment there is no sophisticated algorithm implemented which determines the task with the least progress. The master just splits the task with the highest root vertex in the overall search tree. Another strategy would be to compute the number of childs of the root vertex of a task and to send the split message to the task with the largest number of childs.

In contrast to many other combinatorial optimization problems which are treated with branch-and-bound methods the optimal value of the solutions are known in advance. So we don't have to start with an upper bound and broadcast every new better upper bound to each task.

Each task performs the search loop described in step 5 of algorithm 1. Whenever one branch of the root has been completely enumerated the slave tells the master the actual state of its computation.

After a fixed number of loops the task looks for an incoming split message of the master. This is done in nonblocking mode. If a split message has arrived the task tests if there is a branch from the root vertex which has not been enumerated yet. If such a branch exists, the task spawns a new task. The machine name, where this new task has to be created, is included in the split message of the master. The old task regards the first child of the momentarily enumerated branch of the root as its new root. Every other branch of the old root has to be enumerated by the newly created slave. If there is no further branch of the root to enumerate, the slave doesn't spawn a new task and informs the master that the splitting was not successful.



PVM itself uses a round robin approach to spawn the task. These scheme is inappropriate for our algorithm. We noticed bad behaviour of the algorithm because some machines got to much load. So we were forced to implement the load balancing for ourself. The master now maintains its own list of machines and their respective tasks, so he can supply with each split message an appropriate machine name where to spawn the new task.

The strategy that the master will be informed about every branch of the root that has been enumerated, enables the master to recover from errors in the virtual machine like erasing computers from the virtual machine during the computation. Each task can be started a second time without enumerating solutions twice.

Results

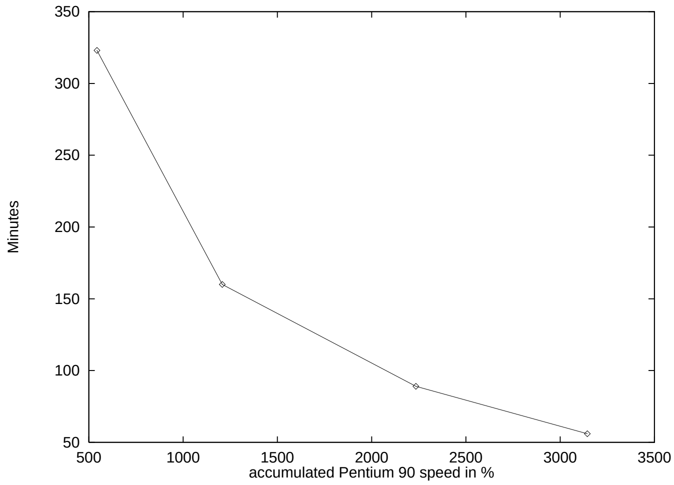
In order to measure the running time each computer in the virtual machine was tested with the serial version of the program computing a small example. The example was the input matrix `KM_PGL_2_5h2_t7_k8`. It is a 54×131 matrix and the dimension of the search space is equal to 79. There are 14 solutions which give 7 nonisomorphic 7-(26,8,6) designs, see [3].

We express the speed of each machine in percentage of the runtime on a Pentium 90 running with Linux. All computers except the DECs used the gcc-compiler. The programs were compiled with the compiler option -O2. Since in general the Gram-Schmidt vectors of an integer basis contain rational numbers, nearly the whole computation time is needed for floating point arithmetics.

internet address	machine type	sec	P90 speed
btm2x4.mat.uni-bayreuth.de	DEC 3000/600 AXP 266 MHz	27	609 %
diva.univ-mlv.fr	HP 9000 755 / 99 MHz	55	303 %
btm2xg.mat.uni-bayreuth.de	DEC 3000/600 AXP 175 MHz	57	294 %
onyx.univ-mlv.fr	Silicon Graphics	67	249 %
weyl.univ-mlv.fr	HP 9000 712 / 80 MHz	70	240 %
btm2xf.mat.uni-bayreuth.de	DEC 3000/400 AXP 130 MHz	78	216 %
free0.univ-mlv.fr	Silicon Graphics	92	181 %
134.245.104.130 (Kiel)	Intel Pentium 90 MHz	167	100 %
btmdx2.mat.uni-bayreuth.de	Intel 486 DX4/100	298	56 %

Several tests with PVM were done with the input matrix `KM_PGL23plus_t6_k8` and $\lambda = 36$. 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 [3]. The following table list the results of our tests of four different configurations. The second column lists the number of computers of each type which were used. The third column contains the types of the computers. The fourth column contains the percentage of Pentium 90 speed which was measured with a small example as it was explained above. In the column "No. Proc." are the maximal number of slave processes 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 in the above table we also found 10008 solutions for the same matrix `KM_PGL23plus_t6_k8` and $\lambda = 45$ which were previously not known to exist. The computing time was 7:34 hours.

Construction of Finite Solvable Groups

Group Extensions

A finite group G is called *solvable*, if there exists a chain of normal subgroups $1 = G_0 < G_1 < \ldots < G_s = G$ such that each group G_i is normal in its succesor G_{i+1} and that the index $[G_{i+1} : G_i]$ is prime for $i = 0, \ldots, s-1$.

Consider the simplest situation of a *group extension* G with a normal subgroup N of prime index $p = [G : N]$ (compare HUPPERT I, 14.8 [22]). Take an arbitrary $g \in G \backslash N$. Then $G = \langle N, g \rangle$ and the factor group G/N consists of the cosets $N, Ng, Ng^2, \ldots, Ng^{p-1}$. Any coset Ng^i with $i \not\equiv 0 \bmod p$ generates the factor group. Because of $(Ng)^p = Ng^p = N$ one gets $g^p = h \in N$. As N is normal in G, $g^{-1}ng = n^g$ is an element of N for any $n \in N$. Conjugation with the fixed element $g \in G$ defines an automorphism on N since $(n_1n_2)^g = g^{-1}n_2gg^{-1}n_2g = n_2^g n_2^g$. This *inner automorphism* of G considered as an automorphism of N is not generally an inner automorphism of N. Define $\alpha_g : N \to N, n \mapsto n^g$, the associated automorphism. What is known about α_g?

i) $g^p = h \in N \cap \langle g \rangle$, a cyclic and therefore abelian subgroup, so $h^g = h^{\alpha_g} = h$.

ii) $g^p = h$ implies for any $n \in N$: $n^{g^p} = n^{\alpha_g^p} = n^h$, so that $\alpha_g^p = \mathrm{inn}_h$, where inn_h is the inner automorphism of N induced by conjugation with h.

On the other hand, one easily verifies that for any group N and any pair of elements $h \in N$ and $\alpha \in \mathrm{Aut}(N)$ there exists a group G of order $p \cdot |N|$, if

i) $h^\alpha = h$ and

ii) $\alpha^p = \mathrm{inn}_h$.

One obtains the group extension by introducing an element g with $g^p := h$ and $g^{-1}ng := n^\alpha$. $\langle N, g \rangle$ defines a group G of order $p \cdot |N|$ and, by definition, N is normal in G.

Elements $h \in N$ and $\alpha \in \mathrm{Aut}(N)$ where N is a fixed group are called *admissible*, if the conditions i) and ii) of above are fullfilled. Let now h and α be an arbitrary admissible pair for a prime p and a group N. Define the group extension $\mathrm{Ext}(N, p, h, \alpha)$ to be the group generated by the elements of N and an element g with $g^p := h$ and $n^g := n^\alpha$ for any $n \in N$. Define the set of all possible group extensions $\mathrm{Ext}(N, p)$ to be $\{\mathrm{Ext}(N, p, h, \alpha) | \, h \in N, \alpha \in \mathrm{Aut}(N), h \,\text{and}\, \alpha \,\text{admissible for}\, N \,\text{and}\, p\}$. For $\mathcal{G}$ a set of groups one defines $\mathrm{Ext}(\mathcal{G}, p) := \cup_{G \in \mathcal{G}} \mathrm{Ext}(G, p)$.

Clearly, for constructing such group extensions a good knowledge of the automorphism group is necessary.

Because subgroups of solvable groups are again solvable it is possible to construct solvable groups by iteration of the procedure just indicated: Assume one wants to find all solvable groups of a given order n. Because the order of subgroups always divide the order n, the lattice of divisors of n comes into the game. Inductively, all subgroups have already been determined and one has to construct all group extensions for all possible subgroups of prime index (in the group of order n). Therefore, consider the prime factorization of n. Any rearrangement of the primes might possibly occure as a sequence of orders of factor groups in the normal chain for a solvable group of order n. So, one starts at the bottom (the trivial group) constructing all groups of prime order. Then one goes up and determines all groups with order a product of two primes, three primes and so on. Note that the lattice of subgroups of a group of order n can be divided into layers according to the number of primes of the corresponding group orders (counting multiplicities).

Denoting the set of solvable groups of order n by $\mathcal{AG}_n$ (German: "auflösbare Gruppe") one has

$$\mathcal{AG}_n = \bigcup_{p|n, p\,\text{prime}} \text{Ext}(\mathcal{AG}_{n/p}, p). \tag{2.8}$$

For the prime power case this becomes the simple case $\mathcal{AG}_{p^k} = \text{Ext}(\mathcal{AG}_{p^{k-1}}, p)$. Because groups of prime power order are solvable, any group of prime power order will be obtained this way.

In the following we will specialize our notation to the case of solvable groups $\mathcal{AG}_n$. Let p and q be primes dividing n. Then – possibly – extensions $G_1 \in \text{Ext}(\mathcal{AG}_{n/p}, p)$ and $G_2 \in \text{Ext}(\mathcal{AG}_{n/q}, q)$ may be isomorphic groups. So, the task is to *find* all the extensions for a given order and afterwards to *reduce* the set of groups up to isomorphism.

As an example, consider the two groups of order 4 and their extensions to groups of order 8:

$4\#1 \simeq Z_2 \times Z_2$:

$$\begin{array}{rcl} A^2 & = & id \\ B^2 & = & id, \quad A^B = B^{-1}AB = A \end{array}$$

$$\begin{array}{cccc} id & A & B & BA \\ A & id & BA & B \\ B & BA & id & A \\ BA & B & A & id \end{array}$$

$4\#2 \simeq Z_4$:

$$\begin{array}{rcl} A^2 & = & id \\ B^2 & = & A, \quad A^B = B^{-1}AB = A \end{array}$$

$$\begin{array}{cccc} id & A & B & BA \\ A & id & BA & B \\ B & BA & A & id \\ BA & B & id & A \end{array}$$

Next, we show the groups together with their subgroup lattice (where only selected cover-relations are drawn). In the text below one can find generators and relations, the column to the right gives a label for the group, the isoclinism class, the order of the first central factor group, the order of the derived subgroup, the order of the automorphism group, their factorization, the Sylow type and the number of conjugacy classes of subgroups with respect to conjugation by the full automorphism group (first line), by the group of only inner automorphisms (middle) and by the trivial group (bottom). So, the last line gives the number of groups in any layer of the lattice. To the right of the closing curely braces the sums of the entries are given. There is not enough space here to explain all this in more details, the interested reader might have a look at:

`http://btm2xd.mat.uni-bayreuth.de/home/research.html`

To define group extensions, it is necessary to study the automorphism groups of 4#1 and 4#2. The group $Z_2 \times Z_2$ admits as an automorphism any permutation of its non-

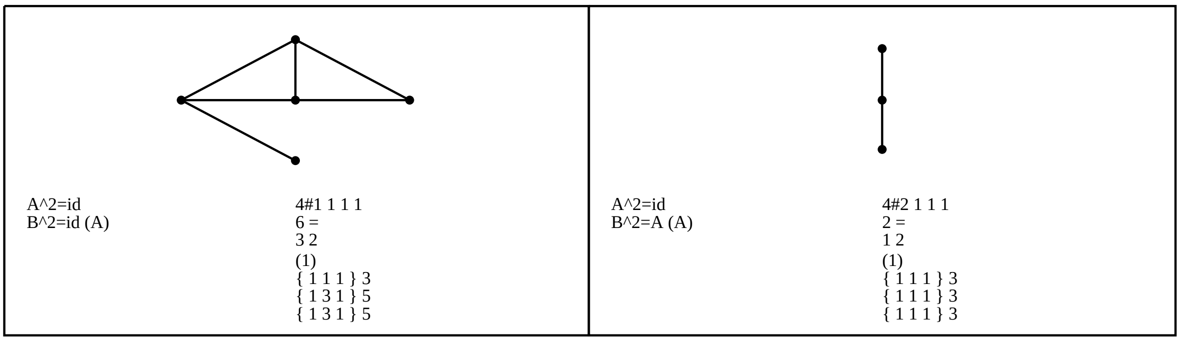


Figure 2.1: The two groups of order 4

trivial elements. So, $\mathrm{Aut}(4\#1) \simeq S_3$. In the other case, there is only one non-trivial automorphism, namely the map $B \mapsto BA = B^{-1}$ (mapping A onto itself). In this case $\mathrm{Aut}(Z_4) \simeq Z_2$.

In the following, we often substitute the elements of the groups by their lexicographic numbers, always counting from 0 on for convenience. Thus we have $0 = id$, $1 = A$, $2 = B$, $3 = BA$ (because $B^2 = id$). As permutation groups, $\mathrm{Aut}(4\#1) = \langle(1\,2),\,(2\,3)\rangle$ and $\mathrm{Aut}(4\#2) = \langle(2\,3)\rangle$.

In order to compute admissible pairs $\alpha \in \mathrm{Aut}(N)$ and $h \in N$ (where $N \in \{4\#1, 4\#2\}$) we define the *extension matrix* of a group N:

$$\mathrm{E}(N,p) = (e_{\alpha,h})_{\alpha\in\mathrm{Aut}(N),h\in N} = \begin{cases} 1 & \Leftrightarrow \quad h^\alpha = h \wedge \alpha^p = \mathrm{inn}_h \\ 0 & \quad\;\; \text{otherwise} \end{cases}, \tag{2.9}$$

i.e. $e_{\alpha,h}$ is 1 iff (α, h) is an admissible pair (for N and p). The elements of $\mathrm{Aut}(4\#1)$ (left column) and $\mathrm{Aut}(4\#2)$ (automorphisms listed by their images on the generators and as permutations of the elements) are:

$$\begin{array}{cllc} \text{no.} & \multicolumn{2}{l}{\alpha \in \mathrm{Aut}(4\#1)} & \mathrm{ord}(\alpha) \\ 0 & [1,2] & id & 1 \\ 1 & [1,3] & (2\,3) & 2 \\ 2 & [2,1] & (1\,2) & 2 \\ 3 & [2,3] & (1\,2\,3) & 3 \\ 4 & [3,1] & (1\,3\,2) & 3 \\ 5 & [3,2] & (1\,3) & 2 \end{array} \qquad \begin{array}{cllc} \text{no.} & \multicolumn{2}{l}{\alpha \in \mathrm{Aut}(4\#2)} & \mathrm{ord}(\alpha) \\ 0 & [1,2] & id & 1 \\ 1 & [1,3] & (2\,3) & 2 \end{array} \tag{2.10}$$

The extension matrices are:

$$E(4\#1,2) = \begin{pmatrix} X\,X\,X\,X \\ X\,X\,.\;\,. \\ X\;.\;\;.\;X \\ .\;\;.\;\;.\;\;. \\ .\;\;.\;\;.\;\;. \\ X\;.\;X\;. \end{pmatrix} \qquad E(4\#2,2) = \begin{pmatrix} X\,X\,X\,X \\ X\,X\,.\;\,. \end{pmatrix} \tag{2.11}$$

So, there are 10 possible extensions of 4#1 and 6 extensions of 4#2. As noted before, one cannot expect 16 different groups of order 8 since some of the candidates might be isomorphic.

By computing Ext(4#1) one gets three non-isomorphic groups (compare figure 2.2): the first is $Z_2 \times Z_2 \times Z_2$ and will be called 8#1. The second is $Z_4 \times Z_2$ (8#2), the third 8#3 is non-abelian: $A^2 = id$, $B^2 = id$, $C^2 = id$ with relations $A^B = A$, $A^C = A$ and $B^C = AB$. This defines a dihedral group and is also an example where computers give other presentations for a group than humans would expect.

In computing Ext(4#2) one obtains the groups 8#2 and 8#3 again. But there are two new groups: 8#4 is cyclic of order 8 and 8#5 is the (non-abelian) quaternionic group: $A^2 = id$, $B^2 = A$, $C^2 = A$, $A^B = A$, $A^C = A$ and $B^C = AB$. It is clear that – for example – one cannot get 8#1 as an extension of $4\#2 \simeq Z_4$: the elementary abelian group has no subgroup isomorphic to Z_4.

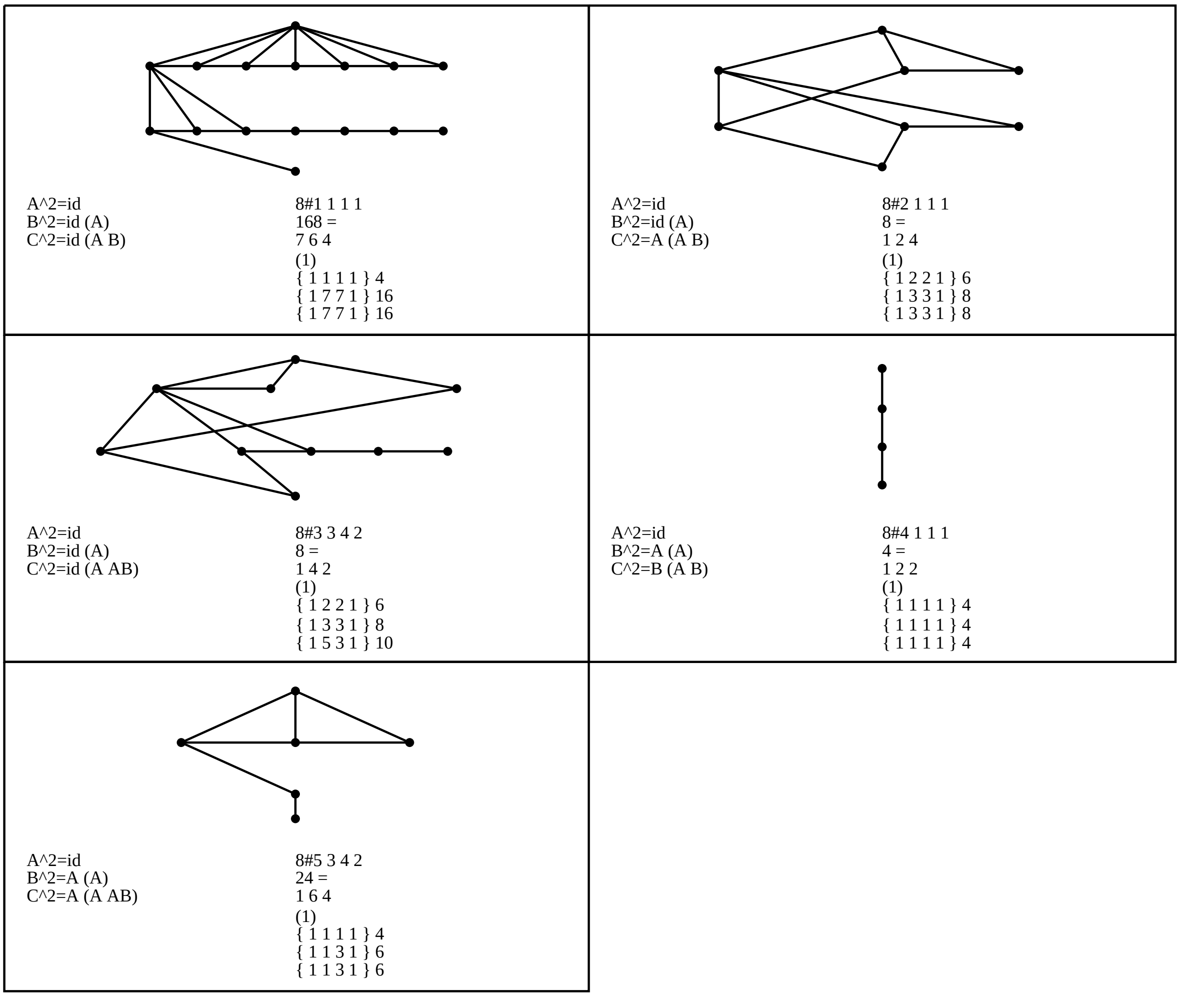


Figure 2.2: The groups of order 8

Reductions

As a general aim, it is desirable to compute a list which is both *complete* and *irredundant* – that is, a representative of each isomorphism type is present and no two groups in the list are of the same isomorphism type. Completeness is guaranteed by the remarks of section 1; irredundancy involves solving the *isomorphism problem* for groups. This is yet another story which definitively cannot be solved in this paper. One would have to talk about invariants of groups and about canonical forms to solve the isomorphism problem.

The amount of computational effort needed for constructing all groups of given order depends much on the number of candidates of groups which have to be tested. Let us try to reduce this number without losing completeness.

The previous example demonstrated the construction of Ext(4#1)∪Ext(4#2) which led to all groups of order 8. But by this process each group was computed too often, because isomorphic copies of the groups were obtained repeatedly. A good idea would be to reduce of the number of extensions, i.e. of the number of entries 1 in the corre-spondig extension matrix. At first, let us start by reducing the number of rows of the extension matrix. Afterwards one reduces the number of entries 1 inside the rows:

i) ("Aut(N) reduction:")
Let $\beta \in \mathrm{Aut}(N)$ be fixed. In the definition of a group extension $\mathrm{Ext}(N,p,h,\alpha) = \langle N, g | g^p = h, n^g = n^\alpha, \forall n \in N\rangle$ one replaces the group N by its image $N^\beta \simeq N$. This gives $g^p = h^\beta$ and conjugation is described by the equation $n^g = n^{\beta\beta^{-1}g\beta} = (n^\beta)^{g^\beta}$. In other words: β extends to an isomorphism $\tilde{\beta} : \mathrm{Ext}(N,p,h,\alpha) \to \mathrm{Ext}(N,p,h^\beta,\alpha^\beta)$. Because one is interested only in non-isomorphic groups, one group extension can be eliminated.

ii) ("Aut(G/N) reduction:")
Let $\gamma : G/N \to G/N, g \mapsto g^j$ with $j \in \{1,\dots,p-1\}$ be an automorphism of G/N. In the group extension $G = \mathrm{Ext}(N,p,h,\alpha)$ one replaces g by $g^\gamma = g^j$ and gets an isomorphic group. In this group $(g^\gamma)^p = g^{jp} = h^j$, and conjugation is determined by $n^{g^\gamma} = n^{g^j} = n^{\alpha^j}$. Thus one of the groups $\mathrm{Ext}(N,p,h,\alpha)$ and $\mathrm{Ext}(N,p,h^j,\alpha^j)$ can be eliminated.

Consider the effect of these two remarks upon the problem of group extensions via extension matrices: according to i) there is a bijection between the entries 1 in rows corresponding to elements α and α^β of $\mathrm{Aut}(N)$ (compare (2.11)) – even more: corresponding group extensions define isomorphic groups. So it is sufficient to generate the extensions just for one representative of each conjugacy class. Let $\mathcal{C}$ denote a system of representatives for the classes of $\mathrm{Aut}(N)$. One only needs to consider the (reduced) extension matrices which have their rows in bijection with the elements of $\mathcal{C}$.

But one can do more: For any $\alpha_0 \in \mathcal{C}$, consider the centralizer $C_G(\alpha_0) = \{\beta \in \mathrm{Aut}(N) | \beta\alpha_0 = \alpha_0\beta\}$. Applying i) to any admissible pair α_0, h one gets the isomorphic group $\mathrm{Ext}(N,p,h^\beta,\alpha_0)$. So, in the α_0-row of the extension matrix one can consider the group action of $C_G(\alpha_0)$ on the entries 1: For each orbit of $C_G(\alpha_0)$ on admissible pairs

$(\alpha_0, h), h \in N$ one chooses exactly one h. All $h^\beta \neq h$ get their 1 replaced by 0. Finally, one gets a *reduced extension matrix.*

Consider reduced extension matrices for the extensions $\mathrm{Ext}(4\#1, 2)$ and $\mathrm{Ext}(4\#2, 2)$. First of all, a list of representatives of the classes of elements of $\mathrm{Aut}(4\#1)$ is needed ($\mathrm{Aut}(4\#2)$ is abelian, so one already knows the representatives):

no.	$\alpha \in \mathrm{Aut}(4\#1)$		$\mathrm{ord}(\alpha)$
0	[1, 2]	*id*	1
1	[1, 2]	(2 3)	2
2	[2, 3]	(1 2 3)	3

(2.12)

The reduced extension matrices are:

$$E(4\#1, 2) = \begin{pmatrix} XX.. \\ XX.. \\ .... \end{pmatrix} \qquad E(4\#2, 2) = \begin{pmatrix} XXX. \\ XX.. \end{pmatrix} \tag{2.13}$$

Altogether, one has only 9 extensions to consider (instead of 16 before) to find all the groups of order 8. Note that remark ii) only gives a reduction for $p \neq 2$.

Parallelization

Let us try to parallelize the computation of $\mathrm{Ext}(\mathcal{N}, p)$. Just before starting, one has to look at the program in order to discover the possible approaches for parallelization. In our case two attempts were made to parallelize at different positions: the crucial point is to know where in the program most of the work is located. One has to recognize subproblems in the algorithm which can be computed widely independent, i.e. without the need of too much communication between the parts. An important concept is the notion of *task granularity* which is the ratio of computation to network activity of a task. It is this parameter which can decide about success or failure of a parallelization in many cases. So, keeping this value in mind is very important.

The serial version of the program acts in the following way (compare figure 2.3): For any group $N \in \mathcal{N}$ all possible p-extensions are computed by the generator (using reduced extension matrices). These are the candidates which have to be filtered up to isomorphism. An important tool for parallelism is a so called "pool of tasks".

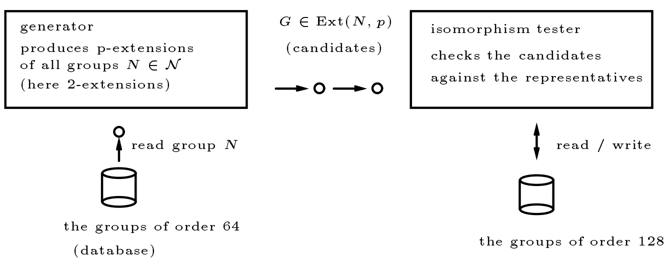


Figure 2.3: The serial program

Here one collects all the subproblems which shall be processed in parallel. A one-to-many relationship between the involved programs is also included into this model: one certain program acts as a controller and administrates the pool (it is therefore called "master"). He also controls the "many" part, namely those programs doing the parallel work (therefore called "slaves") (compare figure 2.4). The "pool of tasks" model can

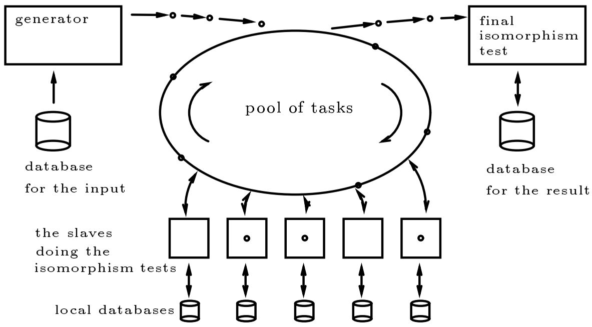


Figure 2.4: The pool of tasks model

be compared with the situation at the airport: The distribution of the luggage to the travellers is solved by circulation; the suitcases (which can be compared with the tasks here) run upon a cyclic band until they are fetched by some traveller. In our language of distributed computing, the task is assigned to a particular slave for processing. But what is a task and what shall the slaves do with it?

There are two approaches. Figure 2.4 just shows the first one by the labels (the model is independent and will also be used in the second algorithm).

The first way is to parallelize the isomorphism part of the problem, i.e. the right hand side of figure 2.3. For any newly generated group it has to be tested whether the group is an isomorphic copy of another group already computed or not. This work is done by the slaves, each of them holding a local database of all groups which already have been computed. The newly generated groups are deposited at the pool of tasks where they reside until some slave is idle (and willing to do the job, just to say it in human language). At that moment such a group is transfered to the slave. If it proves to be isomorphic to one of the groups already in the list it can be skipped (this means a message to the master). In the other case, the group comes back to the master (or only a tag which group is meant because the group is already held at the master). The slave is marked to be idle again. It will get another task if there is any. But what about the group sent back to the master with the information: "non-isomorphic to groups $1, \ldots, l$" where l is the number of groups at the slave's list. Now we see the difficulty: In the meantime, the master might have defined new groups and so further tests are necessary. But maybe there will not be too many new groups, so one can do this instantly. One has to be careful at this point to avoid possible bottlenecks in the computation. Experience shows that the ratio of definition of new groups is relatively low (compared to the number of candidates which have to be tested). So, need for late isomorphism tests at the master will not occure too often. Finally, if the group passes these tests too it is really new: The master adds it to its own list of groups and distributes it to all isomorphism slaves so that they can complete their lists.

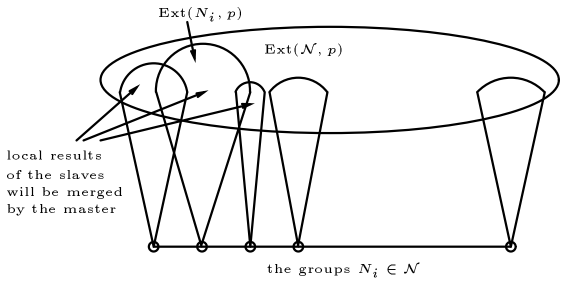


Figure 2.5: The parallel group extension model

A second approach is parallelization according to equation (2.8) (see figure 2.5). Again, one uses a pool of tasks for the distribution of the groups $N \in \mathcal{N}$. Each slave gets his own group for calculating $\mathrm{Ext}(N,p)$. The results of the slaves will be sent back to the master and there merged against each other. One might suppose a parallel merging in form of a binary tree but one has to admit that the results of the slaves will come back in no predictable order: the computation of $\mathrm{Ext}(N)$ may sometimes be very difficult, sometimes very easy (the amount of work depends for instance on the size – or better the number of entries 1 – of the extension matrix). Thus a linear merge was prefered in the actual implementation.

Some remarks should be added concerning the aspect of holding the data in the algorithm. As the number of groups can be quite large for higher n, the amount of storage needed by the slaves should not be under-estimated. At the moment (in the current implementation) each slave holds its own database for ease of access (no collisions during write). One has to admit that the access to common disk space via NFS can cause another bottleneck. Local disks will help here. But since the list of groups is the same for all slaves, an approach of a common database seems to be a good idea. The *PIOUS* [24] system for parallel file IO could support this aspect of the problem. PIOUS is able to use network wide distributed disk space for its files. The total amount of storage would be widely reduced. Maybe the network traffic increases a little. This question is not yet tested but PIOUS seems to be an interesting approach.

Results: Serial vs. Parallel Version

The computation includes for each group:

i) a canonical form producing a (unique) hash key for any group,

ii) the automorphism group,

iii) the conjugacy classes of the automorphism group,

iv) the Sylow type,

v) the subgroup lattice

(a) according to conjugation by *inner* automorphisms

(b) according to conjugation by *outer* automorphisms (using the automorphism group computed already in ii),

vi) the isoclinism class following the definition of isoclinism of Hall and Senior [21].

At first, let us try to compare the speed of different architectures. This seems to be necessary because results of general benchmarks cannot easily be transfered to the specific program which is run here. For the test, we just run a very small example. Note that this program uses only integers (no floating points) and is very special in its kind because it merely does not "compute" in the narrow sense. It is more a collection of lots of (deeply nested) loops. Moreover, the program does not take advantage of DECs 64 bit processors. This might help to explain why hardware based on Intel Pentium is so well suited for the program (considering also prices !).

On each platform, a high grade of optimization was tried (`cxx` supports optimization only up to `-O2`). Results without absolute time were obtained in a previous version of the test (previous version of the program). The ratio of P90 speed should be the same (see table 2.1).

internet address	machine type	cmplr.	h:mm:ss	P90 speed
	PentiumPro 200 MHz	`g++ -O3`	10:52	299 %
btm2x4.mat.uni-bayreuth.de	DEC AlphaStation 600, 333 MHz	`cxx -O2`	11:06	293 %
raptor.mch.sni.de	SGI PowerChallenge chip: R10000, 190 MHz	`g++ -O3`	12:21	263 %
rome.univ-mlv.fr	HP 9000 755 / 99 MHz	`g++ -O2`		116 %
diva.univ-mlv.fr	HP 9000 755 / 99 MHz	`g++ -O2`		106 %
tarry.math.uni-kiel.de	Intel Pentium 90 MHz	`g++ -O3`	32:30	100 %
btrzxa.hrz.uni-bayreuth.de	SGI	`g++ -O3`	34:51	93 %
btm2xg.mat.uni-bayreuth.de	DEC Alpha 3000 / 600, chip 21064, 175 MHz	`cxx -O2`	34:55	93 %
weyl.univ-mlv.fr	HP 9000 712 / 80 MHz	`g++ -O2`		91 %
btrcx4.hrz.uni-bayreuth.de	Silicon Graphics Indy	`g++ -O3`	38:29	84 %
btm2xd.mat.uni-bayreuth.de	IBM RS 6000	`xlC`		71 %
btm2xf.mat.uni-bayreuth.de	DEC Alpha 3000 / 400, chip 21064, 130 MHz	`cxx -O2`	49:49	65 %
	Intel 486 DX2/50 MHz VL	`gcc -O3`	1:36:08	34 %

Table 2.1: The serial version

Testing the PVM version of the program involves some difficulties. It only makes sense to measure the real time of the run, i.e. the life-time of the master. There is no way of mesuring "user-time" as it was done in the serial version. Thus, the load of the computer imposed by other users has a negative impact on the evaluation. The values presented here were obtained during a week-end's night with no other processes running on the machines. It is also desirable to test the PVM program on a pool on homogeneous machines so that one can study the effect of succesively increasing the number of processors. The test runs presented here were made on a pool of (equal) SGI workstations in the computing center of Bayreuth (see figure 2.6).

For an ideal behaviour of the parallelized version, the test was limited to the computation of the sets of extensions $\mathrm{Ext}(G,p)$ (which was the aim of the parallelized version of the program); the second step of merging these lists of groups was left out for the test. Another possible source of friction comes from task granularity. It has already been discussed that the amount of computation should not be too low compared with

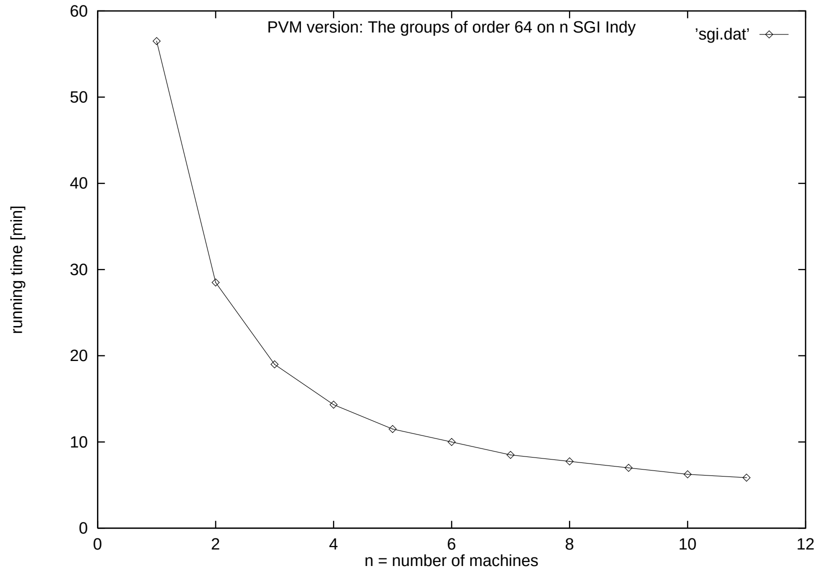


Figure 2.6: Testing the parallel version

the time spent for sending messages and administrating the distributed system. Here we have the other side of the medal: if one task is computing very long, all other tasks might have terminated in between and the computation is unnecesarily lengthened at the end with only one working task. The problem chosen for this particular test was the computation of all groups of order 64 as extensions of the 51 groups of order 32.

Table 2.2 shows the number of solvable groups (nilpotent groups) up to order 127.

n_0+	0	1	2	3	4	5	6	7	8	9
0		1(1)	1(1)	1(1)	2(2)	1(1)	2(1)	1(1)	5(5)	2(2)
10	2(1)	1(1)	5(2)	1(1)	2(1)	1(1)	14(14)	1(1)	5(2)	1(1)
20	5(2)	2(1)	2(1)	1(1)	15(5)	2(2)	2(1)	5(5)	4(2)	1(1)
30	4(1)	1(1)	51(51)	1(1)	2(1)	1(1)	14(4)	1(1)	2(1)	2(1)
40	14(5)	1(1)	6(1)	1(1)	4(2)	2(2)	2(1)	1(1)	52(14)	2(2)
50	5(2)	1(1)	5(2)	1(1)	15(5)	2(1)	13(5)	2(1)	2(1)	1(1)
60	12(2)	1(1)	2(1)	4(2)	267(267)	1(1)	4(1)	1(1)	5(2)	1(1)
70	4(1)	1(1)	50(10)	1(1)	2(1)	3(2)	4(2)	1(1)	6(1)	1(1)
80	52(14)	15(15)	2(1)	1(1)	15(2)	1(1)	2(1)	1(1)	12(5)	1(1)
90	10(2)	1(1)	4(2)	2(1)	2(1)	1(1)	231(51)	1(1)	5(2)	2(2)
100	16(4)	1(1)	4(1)	1(1)	14(5)	2(1)	2(1)	1(1)	45(10)	1(1)
110	6(1)	2(1)	43(14)	1(1)	6(1)	1(1)	5(2)	4(2)	2(1)	1(1)
120	44(5)	2(2)	2(1)	1(1)	4(2)	5(5)	16(2)	1(1)		

Table 2.2: The number of solvable (nilpotent) groups of order $n \leq 127$

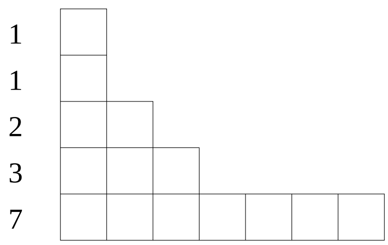
Ribbon tableaux and spin polynomials

Introduction

Ribbon tableaux, introduced by Stanton and White [37] can be used to construct a basis of highest weight vectors in the Fock space representation of the quantum affine algebra $U_q(\widehat{sl}_n)$ [35]. This basis plays a crucial rôle in the computation of the canonical basis of the q-deformed Fock space. It is conjectured in [36] that this basis gives the decomposition matrices of q-Schur algebras at roots of unity. Also, generating functions of ribbon tableaux lead to generalizations of Hall-Littlewood functions, which are q-analogues of products of Schur functions [35].

Ribbon tableaux of given shape and weight can be recursively generated and the corresponding algorithm is well-suited to parallel computing. We have adapted this algorithm in order to compute this set of ribbon tableaux using a parallel virtual machine described by the PVM program. This program mainly returns the generating function of these ribbon tableaux, according to a statistic called spin. We have thus obtained an efficient parallel ribbon tableau generator pushing off the limit of accessible spin polynomials.

A *partition* is a finite sequence $\lambda = (\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r \geq 0)$ of weakly decreasing non-negative integers λ_i, called the *parts* of the partition λ. The sum of all λ_i's is called the *weight* of λ and is denoted by $|\lambda|$. The *length* $l(\lambda)$ of the partition λ is equal to the number of non-zero parts of λ. A partition can be represented by a planar diagram of boxes, called a *Ferrers diagram*: rows of this diagram are numbered bottom-up and the i-th row exactly contains λ_i boxes. For instance, to the partition $\lambda = (7, 3, 2, 1, 1)$ corresponds the following diagram:



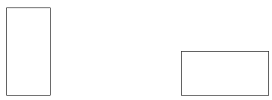
Now, flipping this diagram over its main diagonal (from lower left to upper right) *i.e.* if we choose to read lengths of the columns instead of lengths of the rows, we still obtain a partition λ' called the *conjugate* of λ. In the previous example, we get $\lambda' = (5, 3, 2, 1, 1, 1, 1)$.

A *Young tableau*, or for short a tableau, is a Ferrers diagram (corresponding to a given partition λ) with its boxes filled with positive integers which are weakly increasing across each row – read from left to right – and strictly increasing across each column – read bottom-up. The partition λ is then called the *shape* of the tableau and if we use μ_1 times the letter 1, μ_2 times the letter 2, etc., we say that the *weight* of the tableau is $\mu = (\mu_1, \mu_2, \ldots)$. For example:

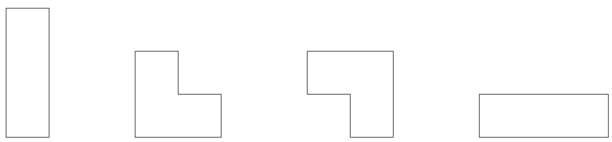
8						
6						
3	5					
2	2	6				
1	1	1	2	4	4	7

is a tableau of shape $\lambda = (7, 3, 2, 1, 1)$ and weight $\mu = (3, 3, 1, 2, 1, 2, 1, 1)$ [34].

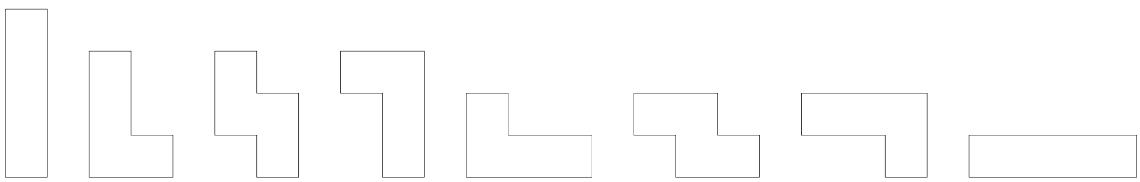
We are now interested in filling a shape – a Ferrers diagram – with *ribbons*. A k-*ribbon* is a connected diagram of boxes – we can go from one box to another one by a path of boxes having one side in common – such that if we run across the ribbon from left to right, we have to go either horizontally or vertically from top to bottom, and such that it does not contain any square 2×2 of boxes. We can easily check that for a given k, we have exactly 2^{k-1} different k-ribbon shapes. For instance, we have 2 dominos for $k = 2$:



4 different 3-ribbon shapes ($k = 3$):



and the following 4-ribbon shapes for $k = 4$:



Not all Ferrers diagrams are tilable with k-ribbons. An obvious necessary condition is that the number of boxes should be divisible by k but this is not sufficient. For instance, the shape $\lambda = (3, 2, 1)$ is not tilable with dominos.

Having filled a given shape λ with some k-ribbons, we associate to each ribbon a positive integer in such a way that integers are weakly increasing across rows from left to right and strictly increasing across columns bottom-up. To make sense of this definition, we have to explain how to read rows and columns. Let the *root* of a ribbon denote the rightmost and lowest box of the ribbon. The exact condition on columns is that the root of a ribbon labeled by the integer i must not be above any box of a ribbon labeled by an integer $j \geq i$. The *weight* of a ribbon tableau is defined as in the case of Young tableaux. For instance, the following picture shows a ribbon tableau filled with 3-ribbons:

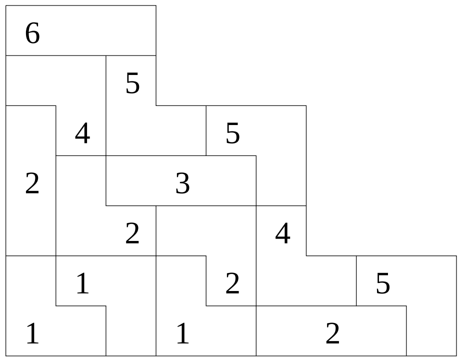
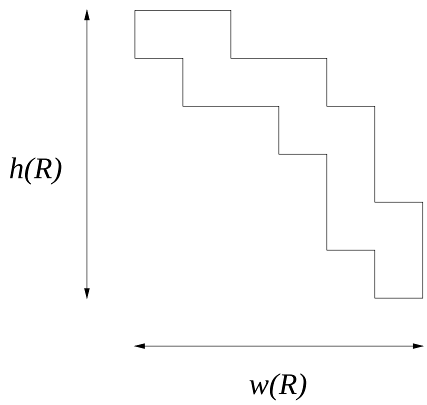


Figure 2.7: Ribbon tableau of shape $\lambda = (9,9,6,6,6,3,3)$ and weight $\mu = (3,4,1,2,3,1)$

Spin Polynomial

We can also define a certain statistic on each ribbon tableau T, called the *spin* of T. Let R be a k-ribbon, $h(R)$ denote its *height* and $w(R)$ its width:



Observe that $h(R) + w(R) = k + 1$. The *spin* $s(R)$ of R is by definition equal to:

$$s(R) := \frac{h(R) - 1}{2} . \tag{2.14}$$

Remark that the spin takes values in $\frac{1}{2}\mathbb{N}$. Now, the *spin* $s(T)$ of a tableau T is set to be:

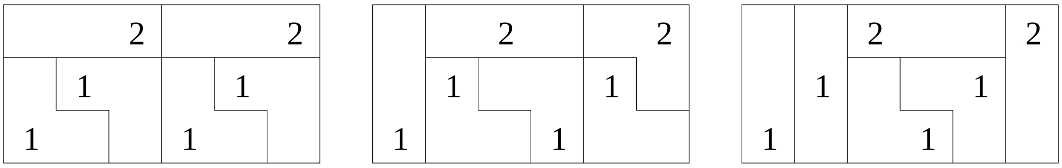
$$s(T) := \sum_R s(R) , \tag{2.15}$$

where the sum is over all ribbons of the tableau. For instance, the ribbon tableau of Fig. 2.7 has a spin equal to 6.

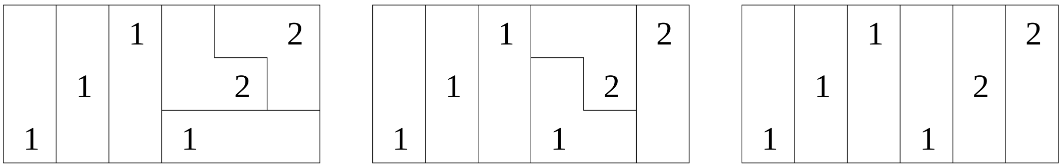
Let λ and μ be two partitions such that $|\lambda| = k|\mu|$ and consider the set $\mathrm{Tab}_k(\lambda, \mu)$ of all ribbon tableaux of shape λ and weight μ composed of k-ribbons. The most important information is the *spin polynomial* $G^{(k)}_{\lambda,\mu}(q)$ defined by:

$$G^{(k)}_{\lambda,\mu}(q) := \sum_{T \in \mathrm{Tab}_k(\lambda,\mu)} q^{s(T)} . \tag{2.16}$$

Example 1 *Let $\lambda = (6,6,6)$ and $\mu = (4,2)$. Then, the set of all ribbon tableaux of shape λ, weight μ, and composed of 3-ribbons is the following:*



2 3 4



4 5 6

in which we have indicated under each tableau its spin. Then, the spin polynomial corresponding to this set of tableaux is $G^{(3)}_{666,42}(q) = q^2 + q^3 + 2q^4 + q^5 + q^6$.

Matrix Coding of Ribbon Tableaux

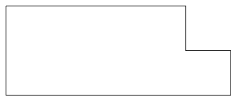
We code a ribbon tableau T of shape λ and weight μ by a matrix $M(T)$ with r rows and s columns, r being the length of the partition μ and s the length of the conjugate partition λ' of λ. This matrix is filled with entries 0 or k, the i-th row (numbered from bottom to top) containing the value k exactly μ_i times. To obtain the i-th row of this matrix, we have to look for the indices of columns containing the root of a ribbon labeled by the integer i. Then, we write k in the corresponding column of the matrix. Other entries are set to be 0. The matrix $M(T)$ that corresponds to Fig. 2.7 is:

$$\begin{array}{c} 6 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{array} \begin{bmatrix} 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 3 & 0 & 0 & 3 \\ 0 & 3 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 3 & 0 & 3 & 0 & 3 & 0 & 0 & 3 & 0 \\ 0 & 3 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

For instance, in Fig. 2.7 ribbons labeled by 2 have their roots in columns $1, 3, 5$ and 8 so that the second row – counting bottom-up – of the matrix is $[3\ 0\ 3\ 0\ 3\ 0\ 0\ 3\ 0]$. To rebuild the ribbon tableau T from its matrix $M(T)$, we first insert the ribbons labeled by 1. The bottom row of the matrix $M(T)$ indicates the positions of roots of ribbons but we have to "fold up" these ribbons. For that purpose, we proceed as follows: assume the matrix has s columns and consider the vector $\rho_s := [s-1, s-2, \ldots, 2, 1, 0]$, then add this vector to the first row – recall that rows are numbered bottom-up – of the matrix. Our example gives:

$$\begin{array}{cccccccccc} & 0 & 3 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \\ + & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \hline = & 8 & 10 & 9 & 5 & 7 & 3 & 2 & 1 & 0 \end{array}$$

Now, we sort this vector in decreasing order, subtract ρ_s and obtain the conjugate of the shape occupied by ribbons labeled by 1. Our example leads to the sorted vector $[10, 9, 8, 7, 5, 3, 2, 1, 0]$ from which we obtain the shape $[2, 2, 2, 2, 1, 0, 0, 0, 0]$ by subtracting ρ_9:

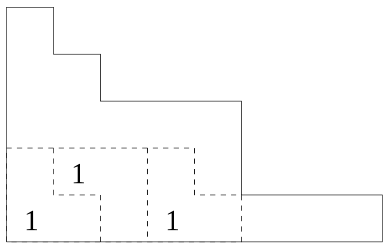


2 2 2 2 1

which is exactly the shape occupied by ribbons labeled by 1 in Fig. 2.7 (we can check that a given shape has at most one tiling with ribbons carrying the same label). To build the shape occupied by ribbons labeled by 2, we start with the sorted vector obtained at the previous step and add the second row in the matrix. Then, we proceed exactly as explained for the first row in order to obtain the shape occupied by ribbons labeled by 1 and 2. As we already know the shape occupied by ribbons labeled by 1, we easily deduce the shape occupied by ribbons labeled by 2. Coming back to our example, to insert ribbons labeled by 2, we have to perform the following operation:

$$\begin{array}{cccccccccc} & 3 & 0 & 3 & 0 & 3 & 0 & 0 & 3 & 0 \\ + & 10 & 9 & 8 & 7 & 5 & 3 & 2 & 1 & 0 \\ \hline = & 13 & 9 & 11 & 7 & 8 & 3 & 2 & 4 & 0 \end{array}$$

The sorted vector is $[13, 11, 9, 8, 7, 4, 3, 2, 0]$ from which we determine the vector $\nu' = [5, 4, 3, 3, 3, 1, 1, 1, 0]$ by subtracting ρ_9. Furthermore, ν' stands for the conjugate of the shape $(8, 5, 5, 2, 1)$ occupied by ribbons labeled by 1 and 2:



5 4 3 3 3 1 1 1

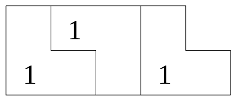
We repeat this construction for ribbons labeled by 3, 4 and so on.

Recursive Computation of Spin Polynomials

This decoding also allows to compute the spin of the corresponding ribbon tableau, at each step of the algorithm. Indeed, having added the vector ρ_s to the first row of the matrix $M(T)$, we obtain a new vector and we have to save the number of inversions I of the permutation that sorts this vector in decreasing order. Our example leads to the vector $[8, 10, 9, 7, 3, 2, 1, 0]$ which is reordered into the vector $[10, 9, 8, 7, 5, 3, 2, 1, 0]$ using three transpositions so that $I = 3$. This number of inversions is exactly equal to:

$$I = \sum_R (w(R) - 1) \ , \tag{2.17}$$

from which we deduce the spin corresponding to the partial following ribbon tableau:

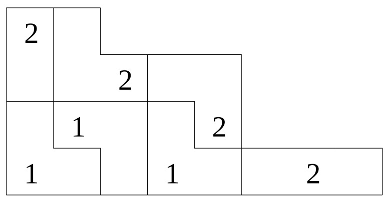


Thanks to (2.14) and (2.17), we have $I = \sum_R (w(R) - 1) = \sum_R (k - h(R))$, which leads to:

$$s(T) = \frac{n(k-1) - I}{2} \ , \tag{2.18}$$

where n is the number of k-ribbons in the partial ribbon tableau T.

To illustrate this point, consider our example in which all ribbons labeled by 1 have a their width equal to 2 so that $I = 3 = (2-1) + (2-1) + (2-1)$ and the spin of the partial ribbon tableau composed of 3-ribbons labeled by 1 is $s(T) = \frac{3}{2}$. Now, the next step is to add ribbons labeled by 2 and obtain the vector $[13, 9, 11, 7, 8, 3, 2, 4, 0]$ that we get sorted into $[13, 11, 9, 8, 7, 4, 3, 2, 0]$ by using 4 transpositions so that $I = 3 + 4$ because I is still equal to the sum $\sum_R (w(R) - 1)$ but this sum is over all ribbons labeled by 1 and 2. The spin of the partial ribbon tableau:



is then equal to $s(T) = \frac{7(3-1)-I}{2} = \frac{7}{2}$.

We are now in position to recursively compute the set $\mathrm{Tab}_k(\lambda, \mu)$ of all k-ribbon tableaux of given shape λ and weight μ together with its corresponding spin polynomial. Indeed, the previous coding of ribbon tableaux by matrices leads to a recursive formula for the following polynomials, from which we deduce the spin polynomials $G^{(k)}_{\lambda,\mu}(q)$. In the following formula, α stands for $\lambda' + \rho_s$ with $s \geq \lambda_1$:

$$F^{(k)}_{\alpha,\mu}(q) := \sum_{T \in \mathrm{Tab}_k(\lambda,\mu)} \Big(\sum_{R \in T} q^{(w(R)-1)} \Big) \ . \tag{2.19}$$

Let $\mu = (\mu_1, \ldots, \mu_r)$. Then, we obtain the possible positions of the μ_r ribbons labeled by r in a ribbon tableau of shape λ and weight μ by reversing the previous decoding algorithm: the global shape containing all ribbons labeled by $1, \ldots, r-1$ are obtained by subtracting from α all distinct permutations of the vector $v_r := [0, \ldots, 0, k, \ldots, k]$

(μ_r components equal k while others equal 0), sorting these vectors in decreasing order and storing the number of inversions. We thus build a tree, rooted by α, with edges indexed by all distinct permutations v_r^σ of v_r and with vertices labeled by the sorted vectors $\alpha - v_r^\sigma$ together with the number of inversions of the sorting permutation.

If one vertex contains a vector with repetitions then it can be shown that this vector cannot belong to a matrix describing a ribbon tableau, so that all edges leading to vertices having repetitions are trivially eliminated. Let $\alpha_1, \ldots, \alpha_m$ denote the remaining vertex labels, and $I_1, \ldots, I_m$ the corresponding numbers of inversions. Let also $\mu^* = (\mu_1, \ldots, \mu_{r-1})$. Then, the recurrence formula is:

$$F^{(k)}_{\alpha,\mu}(q) \;:=\; \sum_{p=1}^{m} q^{I_p}\, F^{(k)}_{\alpha_p,\mu^*}(q) \;\;, \tag{2.20}$$

together with the condition $F^{(k)}_{\rho_s,\emptyset} := 1$. Now, the spin polynomial is given by:

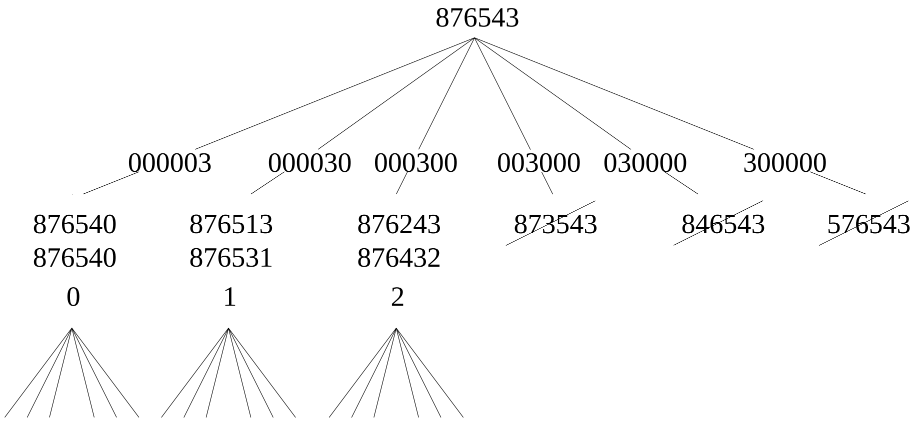
$$G^{(k)}_{\lambda,\mu}(q) \;=\; q^{\frac{|\mu|(k-1)}{2}}\, F^{(k)}_{\alpha,\mu}(q^{-\frac{1}{2}}) \;\;. \tag{2.21}$$

This is actually a polynomial (no fractional exponent) if all parts of λ are divisible by k and this seems to be the most interesting case.

Example 2 *To enumerate all 3-ribbon tableaux of shape $\lambda = (6,6,6)$ and weight $\mu = (4,1,1)$, we have to compute $\lambda' = (3,3,3,3,3,3)$ and take $s = 6$, i.e. the number of columns in the matrix. The vector $\alpha = [3,3,3,3,3,3] + [5,4,3,2,1,0] = [8,7,6,5,4,3]$ is the root of the tree. The number of parts of μ is $r = 3$ so that $v_3 = [0,0,0,0,0,3]$. There are 6 distinct permutations of v_3 among which only three lead to admissible possibilities. We have thus:*

$$F^{(3)}_{876543,411}(q) \;=\; F^{(3)}_{876540,41}(q) + q\, F^{(3)}_{876531,41}(q) + q^2\, F^{(3)}_{876432,41}(q) \;\;,$$

as illustrated by the following graph:



that explicitly gives the first level of the computation. In fact, we get a family of 9 ribbon tableaux, of spin polynomial $G^{(3)}_{666,411}(q) \;=\; q^2 + 2q^3 + 3q^4 + 2q^5 + q^6$.

Parallel implementation

The previous algorithm is able to compute the set of all k-ribbon tableaux of given shape λ and weight μ together with its corresponding spin polynomial $G^{(k)}_{\lambda,\mu}(q)$. We were naturally led to investigate its parallelization and more precisely to study the distribution of the computation of spin polynomials on several computers clustered into a parallel virtual machine described via PVM.

For that purpose, we divide the computation into two different parts, one being managed by the so-called *master* and the other one by the *slaves*, the master and the slaves being tasks in the virtual machine.

On the one hand, the master has to initiate, administrate and terminate the computation of the spin polynomial. Initiate the computation means that it has to read the input data λ, μ and the maximal number of slaves. It also has to administrate the computation of the spin polynomial, *i.e.* to agree or disagree with the requests of creation of new tasks. Then finally, it has to terminate the computation, that is, to give the spin polynomial $G^{(k)}_{\lambda,\mu}(q)$ from the recursively computed polynomial $F^{(k)}_{\alpha,\mu}(q)$ using formula (2.21).

On the other hand, slaves effectively investigate all possibilities. They exactly correspond to the nodes of the previous tree, including the root. Each task has to enumerate all distinct permutations v_r^σ of v_r – that label the vertices – and eliminate the trivially non admissible ones. Then, it asks the master whether it should create new tasks corresponding to the remaining valid vectors, and it generally has to create the accepted new tasks with their corresponding parameters, inform the master that it is dying and finally die.

Now, we have to detail the exact behaviour of the master. As said before, it reads the input data and creates the root of the tree, *i.e.* spawns the first task. Then, it enters in a `while` loop that terminates when there is no more alive task working in the virtual machine. The master is thus waiting for any message from any task. It can be a message informing that a task is dying that corresponds either to a successful result at the lowest level in the tree, or simply to a node that has terminated its work. The master also receives messages carrying requests for new task creations. The answer depends upon several parameters. For instance, the master has to check whether the maximal number of spawned tasks is already reached or not. If this number is already reached then the master informs the task not to spawn new tasks but to recursively compute its corresponding subtree. Futhermore, we can easily check that at a given level of the tree, several sorted vectors may be equal. It follows that the execution tree becomes in fact a graph and that the master has to manage the tasks so that a given subtree is computed only once, even if two – or more – tasks have sent to the master the same request. In any case, this remarks remains true, that is, even if a task is computing in sequential its corresponding subtree, it still sends to the master requests for new task creations or recursive calls. In other words, it is transparent for the master, which only has to count the number of spawned tasks among the total number of tasks. When all tasks are terminated, the master builds the spin polynomial from the partial results obtained from tasks of the lowest level in the execution graph.

Results

The results we obtain are very interesting because they show that the parallelization is useful to efficiently enumerate ribbon tableaux. Previous computations had been done using MAPLE so that it is possible to compare the performance of our program with the results we obtained by using MAPLE. In fact, the execution times of both MAPLE and PVM implementations of the algorithm are not really comparable because the PVM program is a C program which is by itself faster that a MAPLE one. We were able to compute a given polynomial using 8 computers in more or less 1 minute with our PVM program while it took between 2 and 3 hours to compute this given polynomial using our MAPLE implementation of the algorithm. This comparison should be taken as a reference in the sense that we are now able to compute really larger spin polynomials, by enumerating several millions of ribbon tableaux very efficiently.

Our PVM program is called `ribbon`. It allows the user to specify the informations he would like to see at the end of the computation. For that purpose, several options are available among which:

- `data` to print the input data;
- `tasks` to limitate the number of simultaneous spawned tasks;
- `nb` to get the number of solutions;
- `spinpoly` to get the corresponding spin polynomial;
- `labels` to print labels and not only results;
- `execstats` to get execution statistics;
- `graphstats` to get execution graph statistics;
- `maple` to specify that the output format for the spin polynomial is the MAPLE one;
- `tex` to get the spin polynomial as a TeX mathematical formula;
- `symmetrica` to get the spin polynomial as an input readable by the `Symmetrica` program.

For instance, the following lines details a parallel execution involving 8 HP9000 (712, 735 or 755), 7 Silicon Graphics INDY stations, 2 Silicon Graphics INDIGO stations and 4 NEXT stations. The `timex` UNIX command is used to print the execution time statistics of the *master* program but because the execution uses several tasks on several computers we provide a special option (`-execstats`) to administrate the different execution times on each computer and to make statistics on the user time and system time used to compute the solution:

```
$ timex ribbon -tasks 20 -nb -execstats -labels -data
Warning: memory reallocation of level 3
Input data          : 8 12 12 12 12 12 12 12 12 6 4 4 4 4 4 4
Number of solutions : 34990795

Execution statistics:
     total user time: 37.11 sec.
      total sys time: 11.88 sec.
    master user time: 20.60 sec.
     master sys time: 2.21 sec.
     tasks user time: 16.51 sec.
      tasks sys time: 9.67 sec.
      total nb tasks: 2452
       total spawned: 169
   max spawned tasks: 20
        max nb tasks: 1121

real       28.94
user       21.12
sys         2.27
```

This example shows that there are exactly 34990795 ribbon tableaux of shape $\lambda = (12, 12, 12, 12, 12, 12, 12, 12)$ and evaluation $\mu = (4, 4, 4, 4, 4, 4)$. Moreover, the total number of tasks used to compute this result is 2452 among which only 169 tasks has been spawned effectively with a limitation of 20 simultaneous spawned tasks, this limit having been reached during the execution. Furthermore, the maximal number of accepted tasks (in the sense, elementary computations accepted by the master) is 1121 tasks. Here are the same example but with a different number of simultaneous spawned tasks, 10, 15 and 25:

```
$ timex ribbon -tasks 10 -nb -execstats -labels -data
Warning: memory reallocation of level 3
Input data          : 8 12 12 12 12 12 12 12 12 6 4 4 4 4 4 4
Number of solutions : 34990795

Execution statistics:
     total user time: 45.02 sec.
      total sys time: 18.99 sec.
    master user time: 20.00 sec.
     master sys time: 2.38 sec.
     tasks user time: 25.02 sec.
      tasks sys time: 16.61 sec.
      total nb tasks: 2452
       total spawned: 295
   max spawned tasks: 10
        max nb tasks: 698

real       54.06
```

```
user       20.52
sys         2.44

$ timex ribbon -tasks 15 -nb -execstats -labels -data
Warning: memory reallocation of level 3
Input data          : 8 12 12 12 12 12 12 12 12 6 4 4 4 4 4 4
Number of solutions : 34990795

Execution statistics:
     total user time: 32.63 sec.
      total sys time: 12.88 sec.
    master user time: 20.14 sec.
     master sys time: 1.85 sec.
     tasks user time: 12.49 sec.
      tasks sys time: 11.03 sec.
      total nb tasks: 2452
       total spawned: 206
   max spawned tasks: 15
        max nb tasks: 997

real       47.15
user       20.65
sys         1.90

$ timex ribbon -tasks 25 -nb -execstats -labels -data
Warning: memory reallocation of level 3
Input data          : 8 12 12 12 12 12 12 12 12 6 4 4 4 4 4 4
Number of solutions : 34990795

Execution statistics:
     total user time: 38.73 sec.
      total sys time: 8.61 sec.
    master user time: 20.09 sec.
     master sys time: 1.83 sec.
     tasks user time: 18.64 sec.
      tasks sys time: 6.78 sec.
      total nb tasks: 2452
       total spawned: 89
   max spawned tasks: 25
        max nb tasks: 1335

real       26.48
user       20.61
sys         1.88
```

Now we increase the size of the problem by increasing the parts of the shape λ. The computation really needs more resources and the efficiency of our program becomes more and more important. The first example answers that there are 194014660 ribbon tableaux of shape $\lambda = (15, 15, 15, 15, 15, 15, 15, 15)$ and evaluation $\mu = (4, 4, 4, 4, 4, 4)$:

```
$ timex ribbon -tasks 25 -nb -execstats -labels -data
Warning: memory reallocation of level 2
Warning: memory reallocation of level 3
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Input data          : 8 15 15 15 15 15 15 15 15 6 4 4 4 4 4 4
Number of solutions : 194014660

Execution statistics:
     total user time: 166.18 sec.
      total sys time: 28.60 sec.
    master user time: 73.13 sec.
     master sys time: 4.12 sec.
     tasks user time: 93.05 sec.
      tasks sys time: 24.48 sec.
      total nb tasks: 3947
       total spawned: 446
   max spawned tasks: 25
        max nb tasks: 1960

real     1:38.63
user     1:14.75
sys         4.23
```

But let us come to a more interesting example that we have computed on the virtual machine described previously. It deals with the enumeration of all ribbon tableaux of shape $\lambda = (18, 18, 18, 18, 18, 18, 18, 18)$ and evaluation $\mu = (4, 4, 4, 4, 4, 4)$ tiled with 24 $6-$ribbons:

```
$ timex ribbon -tasks 30 -nb -execstats -labels -data -spinpoly -graphstats
Warning: memory reallocation of level 2
Warning: memory reallocation of level 3
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Warning: memory reallocation of level 2
Input data          : 8 18 18 18 18 18 18 18 18 6 4 4 4 4 4 4
Number of solutions : 1079706100
Spin polynomial     : 5*q^6+75*q^7+554*q^8+2835*q^9+
                      11530*q^10+39470*q^11+117927*q^12+314030*q^13+
                      756571*q^14+1665504*q^15+3375529*q^16+6332801*q^17+
                      11046379*q^18+17976562*q^19+27375288*q^20+
                      39109524*q^21+52541166*q^22+66516023*q^23+
                      79511677*q^24+89908914*q^25+96335442*q^26+
                      97959748*q^27+94667092*q^28+87048607*q^29+
                      76241169*q^30+63654091*q^31+50690602*q^32+
```

```
                                  38513325*q^33+27917667*q^34+19299417*q^35+
                                  12713296*q^36+7968919*q^37+4743509*q^38+
                                  2673314*q^39+1420871*q^40+708146*q^41+
                                  328508*q^42+140270*q^43+54314*q^44+18589*q^45+
                                  5424*q^46+1236*q^47+180*q^48

Graph statistics    :
              height: 6
          level info: (1/1) (120/700) (1531/2800) (3006/5600)
                      (1531/2800) (165/700)
     number of nodes: 6354
    max nb daughters: 120 239 211 113 36 1

Execution statistics:
     total user time: 665.67 sec.
      total sys time: 73.99 sec.
    master user time: 277.66 sec.
     master sys time: 7.35 sec.
     tasks user time: 388.01 sec.
      tasks sys time: 66.64 sec.
      total nb tasks: 6353
       total spawned: 1334
   max spawned tasks: 30
        max nb tasks: 2571

real      7:56.93
user      4:42.31
sys          7.56
```

These lines indicate that the spin polynomial is: $180q^{48}+1236q^{47}+5424q^{46}+18589q^{45}+54314q^{44}+140270q^{43}+328508q^{42}+708146q^{41}+1420871q^{40}+2673314q^{39}+4743509q^{38}+7968919q^{37}+12713296q^{36}+19299417q^{35}+27917667q^{34}+38513325q^{33}+50690602q^{32}+63654091q^{31}+76241169q^{30}+87048607q^{29}+94667092q^{28}+97959748q^{27}+96335442q^{26}+89908914q^{25}+79511677q^{24}+66516023q^{23}+52541166q^{22}+39109524q^{21}+27375288q^{20}+17976562q^{19}+11046379q^{18}+6332801q^{17}+3375529q^{16}+1665504q^{15}+756571q^{14}+314030q^{13}+117927q^{12}+39470q^{11}+11530q^{10}+2835q^{9}+554q^{8}+75q^{7}+5q^{6}$ and this polynomial corresponds to more than one billion solutions: 1079706100 distinct ribbon tableaux. Moreover, it gives some details about the execution graph and these informations may be useful to understand the "shape" of the execution graph. For instance, we print a `Warning` each time we need to reallocate memory because we are looking for efficiency. By default, each level of the tree has 700 (in this example) allocated nodes and the master is able to allocate new memory (nodes) when needed. This is notified by a `Warning`. In fact, we can say that 2 reallocations of both levels 2 and 4, and 3 reallocations of level 3 is "quite slight" for generating more than one billion solutions.

The execution graph has 6354 nodes—which correspond to 6453 tasks because the root of the graph is trivial—and is detailled by the two following lines:

```
          level info: (1/1) (120/700) (1531/2800) (3006/5600)
```

```
                      (1531/2800) (165/700)
    max nb daughters: 120 239 211 113 36 1
```

The first one gives, by level, the number of nodes among the number of allocated nodes. The second line indicates the maximal number of successors of a node, for each level. For instance, the second level has at least one node that has 239 possible subtasks to compute. With all these informations, we can imagine the general structure of the graph even if it is not easily drawable.

We have executed several times our program with different parameters, changing essentially the maximal number of spawnable tasks but we obtained more or less the same execution time, *i.e.* the same `total user time`. Our program has also been installed on a network of 3 DEC ALPHA stations (DEC ALPHA 3000 / 600 266 MHz, DEC ALPHA 3000 / 600 175 MHz, DEC ALPHA 3000 / 400 175 MHz) and it gives the following very interesting results:

```
$ time ribbon -tasks 20 -nb -labels -execstats -graphstats -data
Warning: memory reallocation of level 2
Warning: memory reallocation of level 3
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Warning: memory reallocation of level 2
Input data          : 8 18 18 18 18 18 18 18 18 6 4 4 4 4 4 4
Number of solutions : 1079706100

Graph statistics    :
              height: 6
          level info: (1/1) (120/700) (1531/2800) (3006/5600)
                      (1531/2800) (165/700)
     number of nodes: 6354
    max nb daughters: 120 239 211 113 36 1

Execution statistics:
     total user time: 177.01 sec.
      total sys time: 43.32 sec.
    master user time: 56.14 sec.
     master sys time: 3.74 sec.
     tasks user time: 120.87 sec.
      tasks sys time: 39.58 sec.
      total nb tasks: 6353
       total spawned: 1213
   max spawned tasks: 20
        max nb tasks: 1949
57.23u 3.74s 3:56 25% 0+260k 0+1039io 0pf+0w
```

We have also tried to use only one computer, that is the DEC ALPHA 3000 / 600 266 MHz, with one giga-byte of memory and the result was:

```
$ time ribbon -tasks 20 -nb -labels -execstats -graphstats -data
Warning: memory reallocation of level 2
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Warning: memory reallocation of level 3
Warning: memory reallocation of level 4
Warning: memory reallocation of level 3
Warning: memory reallocation of level 2
Input data          : 8 18 18 18 18 18 18 18 18 6 4 4 4 4 4 4
Number of solutions : 1079706100

Graph statistics    :
              height: 6
          level info: (1/1) (120/700) (1531/2800) (3006/5600)
                      (1531/2800) (165/700)
     number of nodes: 6354
    max nb daughters: 120 239 211 113 36 1

Execution statistics:
     total user time: 114.71 sec.
      total sys time: 24.33 sec.
    master user time: 57.23 sec.
     master sys time: 3.89 sec.
     tasks user time: 57.48 sec.
      tasks sys time: 20.44 sec.
      total nb tasks: 6353
       total spawned: 1209
   max spawned tasks: 20
        max nb tasks: 1971

58.32u 3.89s 6:13 16% 0+260k 0+3278io 0pf+0w
```

References

[1] A.Geist, A.Beguelin, J.Dongarra, W.Jiang, R.Manchek, V.Sunderam, *PVM: Parallel Virtual Machine, A users guide and tutorial for networked parallel computing*, MIT Press, 1994.

Integer Linear Equations

[2] 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.

[3] A. Betten, R. Laue, A. Wassermann: Simple 7-Designs With Small Parameters, Spetses, 1996.

[4] E. F. Brickell: Solving low density knapsacks. *Advances in Cryptology, Proceedings of Crypto '83*, Plenum Press, New York (1984), 25–37.

[5] M. J. Coster, B. A. LaMacchia, A. M. Odlyzko, C. P. Schnorr: An improved low-density subset sum algorithm. *Proceedings EUROCRYPT '91, Brighton, May 1991* in *Springer Lecture Notes in Computer Science* **547** (1991), 54–67.

[6] M. R. Garey, D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company (1979).

[7] A. Joux, J. Stern: Improving the Critical Density of the Lagarias-Odlyzko Attack Against Subset Sum Problems. *Proceedings of Fundamentals of Computation Theory 91* in *Lecture Notes in Computer Science* **529** (1991), 258–264.

[8] E. S. Kramer, D. M. Mesner: t-designs on hypergraphs. *Discrete Math.* **15** (1976), 263–296.

[9] D. L. Kreher, S. P. Radziszowski: Finding Simple t-Designs by Using Basis Reduction. *Congressus Numerantium* **55** (1986), 235–244.

[10] M. Kaib, H. Ritter: Block Reduction for Arbitrary Norms. Preprint 1995.

[11] J. C. Lagarias, A. M. Odlyzko: Solving low-density subset sum problems. *J. Assoc. Comp. Mach.* **32** (1985), 229–246.

[12] A. K. Lenstra, H. W. Lenstra Jr., L. Lovász: Factoring Polynomials with Rational Coefficients, *Math. Ann.* **261** (1982), 515–534.

[13] G. L. Nemhauser, L. A. Wolsey: *Integer and Combinatorial Optimization.* John Wiley & Sons (1988).

[14] 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.

[15] C. P. Schnorr: A hierachy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science* **53** (1987), 201–224.

[16] C. P. Schnorr: A More Efficient Algorithm for Lattice Basis Reduction. *J. Algorithms* **9** (1988), 47–62.

[17] C. P. Schnorr, M. Euchner: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Proceedings of Fundamentals of Computation Theory 91* in *Lecture Notes in Computer Science* **529** (1991), 68–85.

[18] A. Wassermann: Finding Simple t-Designs with Enumeration Techniques, in preparation.

Construction of solvable groups

[19] Bettina Eick: Charakterisierung und Konstruktion von Frattinigruppen mit Anwendung in der Konstruktion endlicher Gruppen. Thesis, RWTH Aachen, 1996.

[20] Philip Hall: Classification of prime power groups. *J. für die reine u. ang. Math.*, **182** (1940), pp. 130-141.

[21] Marshall Hall, James K. Senior: The groups of order 2^n $(n \leq 6)$. MacMillan Company, New York, London 1964.

[22] Bertram Huppert: Endliche Gruppen I. Springer Verlag, Berlin, Heidelberg, New York, 1967.

[23] Reinhard Laue: Zur Konstruktion und Klassifikation endlicher auflösbarer Gruppen. *Bayreuther Math. Schr.* **9** (1982).

[24] Steven A. Moyer, V. S. Sunderam: PIOUS for PVM, Version 1.2, User's Guide and Reference Manual. `http://www.mathcs.emory.edu/Research/Pious.html`

[25] Joachim Neubüser: Untersuchungen des Untergruppenverbandes endlicher Gruppen auf einer programmgesteuerten elektronischen Dualmaschine. *Numerische Mathematik* **2**, (1960), 280-292.

[26] Joachim Neubüser: Die Untergruppenverbände der Gruppen der Ordnungen ≤ 100 mit Ausnahme der Ordnungen 64 und 96. Thesis, Kiel 1967.

[27] Edward Anthony O'Brien: The groups of order dividing 256. Thesis, Australian National University, March 1988.

[28] Rodney James, M.F. Newman, Edward Anthony O'Brien: The groups of order 128. *J. Algebra*, **129** (1990), 136-158.

[29] Edward Anthony O'Brien: The p-group generation algorithm. *J. Symbolic Comput.*, **9** (1990), 677-698.

[30] Edward Anthony O'Brien: The groups of order 256. *J. Algebra*, **143** (1991), 219-235.

Ribbon tableaux

[31] C. Carré, B. Leclerc, *Splitting the square of a Schur function into its symmetric and antisymmetric parts*, J. Alg. Comb. **4** (1995), 201–231.

[32] J. Désarménien, B. Leclerc, J.-Y. Thibon, *Hall-Littlewood functions and Kostka-Foulkes polynomials in representation theory*, Séminaire Lotharingien de Combinatoire, Université de Strasbourg, 1993.

[33] A.N. Kirillov, A. Lascoux, B. Leclerc, J.-Y.Thibon, *Séries génératrices pour les tableaux de dominos*, C.R. Acad. Sci. Paris **318** (1994), 395–400.

[34] D. Knuth, *The art of computer programming*, Vol. **3**, Addison-Wesley, 1981.

[35] A. Lascoux, B. Leclerc, J.-Y. Thibon, *Ribbons tableaux, Hall-Littlewood functions, Quantum affine algebras and unipotent varieties*, preprint IGM **96.2**, q-alg/9512031.

[36] B. Leclerc, J.-Y. Thibon, *Canonical Bases of q-Deformed Fock Spaces*, International Math. Research Notices, to appear.

[37] D. Stanton, D. White, *A Schensted algorithm for rim-hook tableaux*, J. Comb. Theory A **40** (1985), 211–247.

Anton Betten
Universität Bayreuth,
Lehrstuhl Mathematik II,
Universitätsstr. 30,
95447 Bayreuth, Deutschland.
Email: `anton@btm2xf.mat.uni-bayreuth.de`
URL: `http://btm2xd.mat.uni-bayreuth.de/betten/anton.html`

Sébastien Veigneau
Institut Gaspard Monge,
Université de Marne-la-Vallée,
2, rue de la Butte Verte 93166 Noisy-le-Grand Cedex, France.
Email: `Sebastien.Veigneau@univ-mlv.fr`
URL: `http://www-igm.univ-mlv.fr/~veigneau/`

Alfred Wassermann
Universität Bayreuth,
Lehrstuhl für Mathematik und ihre Didaktik,
Universitätsstr. 30,
95447 Bayreuth, Deutschland.
Email: `Alfred.Wassermann@uni-bayreuth.de`
URL: `http://did.mat.uni-bayreuth.de/wassermann/wassermann.html`