ANNEXE D : galerie d'exemples

Cette partie est un petit tour d'horizon des codes disponibles sur le Web. Je n'ai donc pas pu inclure certains codes écrits, comme par exemples ceux de M. Pierre Audibert. Le but est de montrer que le domaine des LGA est actif mais pas au point de voir apparaître un consensus ou des convergences sur tous les aspects théoriques ou de programmation. Il est vrai que chaque utilisateur a des besoins particuliers mais beaucoup d'énergie a été perdue à réinventer la même chose à chaque fois. Nous verrons aussi les points forts et faibles des différentes approches.

Les parties de codes présentées ici ne dévoilent qu'une partie des problèmes rencontrés lors du codage de tels programmes. Par exemple, la gestion des "plans temporaires" est souvent passée sous silence, souvent pour des raisons de facilité de programmation au détriment de l'occupation et de la bande passante de la mémoire.

D.1 : Sous X11 :

Ce code a été adapté de celui de Benjamin Temko (btemko@wildspitze.extreme.indiana.edu).
Les routines d'interface ont été changées (appels directs à la Xlib ou aux routines de windef.h) et la boucle principale a été modifiée pour effectuer 1000 itérations afin de mesurer la performance.

Les performances sont honorables mais la partie en virgule flottante est instable sur un Alpha.
En plus, le code est limité à des données sur 32 bits et la taille du tunnel est définie dans le code, ce qui rend son portage délicat et la montée en puissance incertaine. Enfin, comme les particules elles−mêmes ne sont pas représentées, on peut difficilement juger de l'exactitude des règles de collision.

Toutefois c'est le seul code disponible à l'heure actuelle qui soit compréhensible et qui ait de bonnes performances avec un PC sous Linux. Il devra être adapté profondément, en tenant compte des travaux de ce mémoire (strip mining, environnement, listes de modifications, entiers longs etc), pour être accéléré s'il doit faire partie d'un logiciel plus sophistiqué.

type de structure : "vectorielle", 8 plans séparés, donc pas de traitement pair/impair mais beaucoup de pointeurs
collisions : apparemment FHP saturé

Ce programme a été testé dans les conditions suivantes :

```
tunnel : 1024*640 pixels
1000 iterations, 3 affichages
Pentium 200MHz MMX, 32MB SDRAM, 256KO L2
login en root pour effectuer la commande nice --20

compilation:
$ gcc -I/usr/X11R6/include -lX11 -L/usr/X11R6/lib -O3 fhp_0.c

run :
$ nice --20 time -o perf ./a.out

   58.76 user
    0.44 system
 1:00.87 elapsed
     97% CPU
(0 avgtext+0avgdata 0maxresident) k0 inputs+0outputs
 (153major+178minor)page faults
 0 swaps

Il faut 1 minute pour calculer 640*1k*1k=640Mc, le programme traite
donc approximativement 10Mc/s sur ce système.
```

```
reproduit avec permission. (C) 1992 Benjamin Temko

#include "windef.h" /* YG: windef.h est une librairie graphique personnelle,
qui permet d'initialiser l'environnement X, d'afficher les fenêtres et gérer quelques
paramètres de base. */

#define Ranf() (double)((Ranfseed=(Ranfseed*1629+1)%1048576)/1048576.)
#define Ranfset(l) (Ranfseed=(((abs(l)%1048576)*1629)+1)%1048576)
#define SQRT3O2 .8660254
#define SQRT3O4 .4330127
#define SQRT3T2 3.4641016
#define RHO  1.086
#define V  .24
#define STEPSIZE 255
#define XSIZ 1024    /* The X Size of the Automaton*/
#define YSIZ 20    /* The Y Size of the Automaton/32 */
#define GRAN 16    /* coarse graining constant */
```

```
#define XRES 64   /* XSIZ/GRAN */
#define YRES 40  /* YSIZ*(32/GRAN) */
#define RADIUS 60 /* the radius of the cylinder obstacle */

/* A simple vector data type */
typedef struct {
  float Vx, Vy;
}node;

/* Another simple vector data type */
typedef struct {
  short int x, y;
}point;

void nextstate();
void collisions();
void movea();
void moveb();
void movec();
void moved();
void movee();
void movef();
void calcavgs();
void drawit();
void init();
void drawobst();
void inject();

int count=0;
float us=(float) XSIZ*(YSIZ*32);

/* These are the arrays which hold the 8 different particle types */
unsigned int a[XSIZ*YSIZ];
unsigned int b[XSIZ*YSIZ];
unsigned int c[XSIZ*YSIZ];
unsigned int d[XSIZ*YSIZ];
unsigned int e[XSIZ*YSIZ];
unsigned int f[XSIZ*YSIZ];
unsigned int r[XSIZ*YSIZ];
unsigned int s[XSIZ*YSIZ];

/* X11 stuffs */
Window win_main;
GC gc;
XEvent event;

long Ranfseed=42;
point linesize;
int xsyze=1024, ysyze=640;
node result[YRES][XRES];

float proba;
float probb;
float probc;
float probd;
float probe;
float probr;
float probf;

void main(int argc, char **argv)
{
  /* Set up the probabliities for injections */
  proba=(RHO/7.0)+(V/SQRT3T2);
  probb=proba;
  probc=RHO/7.0;
  probd=(RHO/7.0)-(V/SQRT3T2);
  probe=probd;
  probf=probc;
  probr=RHO/7.0;

  /* This is graphics related stuff */
  linesize.x=xsyze/XRES;
  linesize.y=ysyze/YRES;
  /* BDT_begin("FLUID FLOW", xsyze, ysyze, argc, argv); */
```

```
  init_X();
  open_window(main,xsyze, ysyze,black,white,"FLUID FLOW");

  init();
  calcavgs();
  drawobst();
  drawit();

  /* The main loop − inject, calculate, update, repeat! */
  while(count<1000) {
    inject();
    nextstate();
    if(!(++count) {
      calcavgs();
      drawobst();
      drawit();
      printf("%d\n", count);
    }
  }

  close_window(main);
  exitX();
}

/* Draws our obstacle, reading from the s array, which is the STOP bit array */
void drawobst()
{
  int i, j/*, q*/;

  XClearWindow(dpy,win_main);

  for(i=0; i<XSIZ; i++) {
    for(j=0; j<YSIZ*32; j++) {
      if((s[i*YSIZ+(j/32)]>>(j%32))%2 == 1)
        XDrawPoint(dpy,win_main,gc,i,j);
/* BDT_point(i, j); utiliser XDrawPoints() plus tard pour accélérer les appels systèmes  */
    }
  }
}

/* Draws the vectors reading from the vector array */
void drawit()
{
  int x, y;
  point p1, p2;
  for(y=0; y<YRES; y++) {
    for(x=0; x<XRES; x++) {
      p1.x=x*linesize.x;
      p1.y=y*linesize.y+5;
      p2.x=(int)(p1.x+(result[y][x].Vx*(V*100)));
      p2.y=(int)(p1.y+(result[y][x].Vy*(V*100)));
      XDrawLine(dpy,win_main,gc,p1.x,p1.y,p2.x,p2.y);
      /*      BDT_line(p1, p2);                       idem avec XDrawLines()
              BDT_arc(p2, 3, 3, 0, 360); */
    }
  }
}

/* Initialize the arrays and place the obstacle */
void init()
{
  int x, y/*, i, j*/;

  for(y=0; y<YSIZ*XSIZ; y++) {
    for(x=0; x<=31; x++) {
      if(Ranf()< proba) a[y]|=1<<x;
      if(Ranf()< probb) b[y]|=1<<x;
      if(Ranf()< probc) c[y]|=1<<x;
      if(Ranf()< probd) d[y]|=1<<x;
      if(Ranf()< probe) e[y]|=1<<x;
      if(Ranf()< probf) f[y]|=1<<x;
      if(Ranf()< probr) r[y]|=1<<x;
    }
    s[y]=0;
```

```
  }

  for(y=0; y<YRES; y++) {
    for(x=0; x<XRES; x++) {
      result[y][x].Vx=0.0;
      result[y][x].Vy=0.0;
    }
  }

  for(y=256*YSIZ+7; y<256*YSIZ+12; y++) {
    for(x=0; x<=31; x++) s[y]|=(1<<x);
  }
}

/* Inject more particles in from the left. This keeps the flow moving */
void inject()
{
  int i, x;

  for(i=0; i<YSIZ; i++) {
    a[i]=b[i]=d[(XSIZ-1)*YSIZ+i]=e[(XSIZ-1)*YSIZ+i]=0;
    for(x=0; x<=31; x++) {
      if(Ranf()< proba) a[i]|=1<<x;
      if(Ranf()< probb) b[i]|=1<<x;
      if(Ranf()< probd) d[(XSIZ-1)*YSIZ+i]|=1<<x;
      if(Ranf()< probe) e[(XSIZ-1)*YSIZ+i]|=1<<x;
    }
  }
}

/* Calculate the collisions and move the particles */
/* These two actions together make up the state transition */
void nextstate()
{
  collisions();
  movea(); /* cette partie-là perd du temps pour rien : */
  moveb(); /* il aurait mieux fallu tout mettre ensemble */
  movec(); /* au lieu de perdre du temps dans les prologues */
  moved(); /* et les épilogues de fonctions ! ça aurait */
  movee(); /* même permis de fusionner quelques boucles... */
  movef();
}

/* Perform the collisions */
void collisions()
{
  register unsigned int j, nots, ss, aa, bb, cc, dd, ee, ff, rr;
  unsigned int triple;
  unsigned int ad, be, cf;
  unsigned int newad, newbe, newcf;
  unsigned int adb1, adb2, beb1, beb2, cfb1, cfb2;
  unsigned int nad, nbe, ncf;
  unsigned int newac, newbd, newce, newdf, newea, newfb;
  unsigned int a1, b1, c1, d1, e1, f1;
  /*  unsigned int t1, t2, t3;*/
  unsigned int cha, chb, chc, chd, che, chf, chr;
  unsigned int eps, noteps;
  /*  int rpcc=0;*/
  eps=(Ranf()<.5) ? 0 : 0xffffffff;
  noteps=~eps;
  for(j=0; j<=YSIZ*XSIZ-1-1; j++) {
    aa=a[j];
    bb=b[j];
    cc=c[j];
    dd=d[j];
    ee=e[j];
    ff=f[j];
    rr=r[j];
    ss=s[j];
    nots=~ss;
    ad=aa~(bb|cc|ee|ff|rr);
    be=bb~(aa|cc|dd|ff|rr);
    cf=cc~(aa|bb|dd|ee|rr);
```

```
    newad=(eps | (noteps;
    newbe=(eps | (noteps;
    newcf=(eps | (noteps;

    adb1=(bb(ff^cc)(aa|dd|rr));
    adb2=(cc(ee^bb)(aa|dd|rr));
    beb1=(aa(ff^cc)(bb|ee|rr));
    beb2=(cc(aa^dd)(bb|ee|rr));
    cfb1=(aa(ee^bb)(cc|ff|rr));
    cfb2=(bb(aa^dd)(cc|ff|rr));
    nad=adb1|adb2|beb1|cfb1;
    nbe=beb1|beb2|adb1|cfb2;
    ncf=cfb1|cfb2|adb2|beb2;

    newac=bb~(aa|cc|dd|ee|ff);
    newbd=cc~(aa|bb|dd|ee|ff);
    newce=dd~(aa|bb|cc|ee|ff);
    newdf=ee~(aa|bb|cc|dd|ff);
    newea=ff~(aa|bb|cc|dd|ee);
    newfb=aa~(bb|cc|dd|ee|ff);

    a1=ff~(aa|cc|dd|ee|rr);
    b1=aa~(bb|dd|ee|ff|rr);
    c1=bb~(aa|cc|ee|ff|rr);
    d1=cc~(aa|bb|dd|ff|rr);
    e1=dd~(aa|bb|cc|ee|rr);
    f1=ee~(bb|cc|dd|ff|rr);

    triple=(aa^bb)bb^cc)cc^dd)dd^ee)ee^ff)~rr;

    cha=nad|triple|ad|newad|newac|newea|newfb|a1|b1|f1;
    chb=nbe|triple|be|newbe|newbd|newfb|newac|b1|c1|a1;
    chc=ncf|triple|cf|newcf|newce|newac|newbd|c1|d1|b1;
    chd=nad|triple|ad|newad|newdf|newbd|newce|d1|e1|c1;
    che=nbe|triple|be|newbe|newea|newce|newdf|e1|f1|d1;
    chf=ncf|triple|cf|newcf|newfb|newdf|newea|f1|a1|e1;
    chr=a1|b1|c1|d1|e1|f1|newac|newbd|newce|newdf|newea|newfb;

    a[j]=((aa^cha) nots) | (dd;
    b[j]=((bb^chb) nots) | (ee;
    c[j]=((cc^chc) nots) | (ff;
    d[j]=((dd^chd) nots) | (aa;
    e[j]=((ee^che) nots) | (bb;
    f[j]=((ff^chf) nots) | (cc;
    r[j]=rr^chr;
  }
}

/* Move the a particles to their next state */
void movea()
{
  unsigned int /*i,*/ *p, ny2, *pstart, *pend/*, temp[YSIZ]*/;
  register unsigned int *pc;

  pstart=a+(XSIZ-1)*YSIZ;
  pend=a;
  ny2=2*YSIZ;

  for(p=pstart; p>(pend+YSIZ); p-=ny2) {
    for(pc=p; pc<p+YSIZ; pc++)
      *(pc)= *(pc-YSIZ);
    for(pc=p-YSIZ; pc<p-1; pc++)
      *pc= (*(pc-YSIZ) >> 1) | (*(pc-YSIZ+1) << 31);
    *(p-1) = (*(p-YSIZ-1) >> 1 ) | (*(p-ny2) << 31);
  }

  for(pc=a+YSIZ; pc<a+ny2; pc++)
    *(pc)=*(pc-YSIZ);
}

/* Move the b particles to their next state */
void moveb()
{
  unsigned int /*i, */*p, ny2, *pstart, *pend/*, temp[YSIZ]*/;
```

```
  register unsigned int *pc;

  pstart=b+(XSIZ-1)*YSIZ;
  pend=b;
  ny2=2*YSIZ;

  for(p=pstart; p>pend+YSIZ; p-=ny2) {
    for(pc=p+YSIZ-1; pc>p; pc--)
      *pc= (*(pc-YSIZ) << 1) | (*(pc-YSIZ-1) >> 31);
    *p=(*(p-YSIZ) << 1) | (*(p-1) >> 31);
    for(pc=p-YSIZ; pc<=p-1; pc++)
      *pc= *(pc-YSIZ);
  }

  for(pc=b+ny2-1; pc>b+YSIZ; pc--)
    *pc = (*(pc-YSIZ) <<1) | (*(pc-YSIZ-1) >> 31);
  *(b+YSIZ)=(*b << 1) | (*(b+YSIZ-1) >> 31);
}

/* Move the c particles to their next state */
void movec()
{
  unsigned int /*i,*/ *p, *pstart, *pend, temp;
  register unsigned int *pc;

  pstart=c+(XSIZ-1)*YSIZ;
  pend=c;

  for(p=pstart; p>=pend; p-=YSIZ) {
    temp=*(p+YSIZ-1);
    for(pc=p+YSIZ-1; pc>p; pc--)
      *(pc)= (*(pc)<<1 | *(pc-1)>>31);
    *(p)=(*(p)<<1 | temp >> 31);
  }
}

/* Move the d particles to their next state */
void moved()
{
  unsigned int /*i, */ *p, ny2, *pstart, *pend/*, temp[YSIZ]*/;
  register unsigned int *pc;

  pend=d+(XSIZ-1)*YSIZ;
  pstart=d;
  ny2=2*YSIZ;

  for(p=pstart; p<pend-YSIZ; p+=ny2) {
    for(pc=p; pc<p+YSIZ; pc++)
      *(pc)= *(pc+YSIZ);
    for(pc=p+ny2-1; pc>p+YSIZ; pc--)
      *pc= (*(pc+YSIZ) << 1) | (*(pc+YSIZ-1) >> 31);
    *(p+YSIZ)=(*(p+ny2) <<1) | (*(p+ny2+YSIZ-1) >> 31);
  }
  for(pc=pend-YSIZ; pc<pend; pc++)
    *(pc)=*(pc+YSIZ);
}

/* Move the e particles to their next state */
void movee()
{
  unsigned int /* i,*/ *p, ny2, *pstart, *pend/*, temp[YSIZ]*/;
  register unsigned int *pc;

  pend=e+(XSIZ-1)*YSIZ;
  pstart=e;
  ny2=2*YSIZ;

  for(p=pstart; p<pend-YSIZ; p+=ny2) {
    for(pc=p; pc<p+YSIZ-1; pc++)
      *(pc)= (*(pc+YSIZ) >> 1) | (*(pc+YSIZ+1) << 31);
    *(p+YSIZ-1)=(*(p+ny2-1) >>1) | (*(p+YSIZ) << 31);
    for(pc=p+YSIZ; pc<p+ny2; pc++)
      *(pc)= *(pc+YSIZ);
  }
```

```
  for(pc=pend-YSIZ; pc<pend-1; pc++)
    *(pc)= (*(pc+YSIZ) >> 1) | (*(pc+YSIZ+1) << 31);
  *(pend-1)=(*(pend+YSIZ-1) >>1) | (*(pend) << 31);
}

/* Move the f particles to their next state */
void movef()
{
  unsigned int /*i,*/ *p, *pstart, *pend, temp;
  register unsigned int *pc;

  pend=f+(XSIZ-1)*YSIZ;
  pstart=f;

  for(p=pstart; p<=pend; p+=YSIZ) {
    temp=*(p);
    for(pc=p; pc<p+YSIZ-1; pc++)
      *(pc)= *(pc)>>1 | *(pc+1)<<31;
    *(p+YSIZ-1)=*(p+YSIZ-1)>>1 | temp<<31;
  }
}

/* Calculate interesting statistics about the flow */
void calcavgs()
{
  int i, j, ii, /*jj,*/ k, q;
  int cts[7];
  node total;
  int ac,bc,cc,dc,ec,fc,t1=0;
  int xcount=0, ycount=0;
  int statcount=0;
  int gs=GRAN*GRAN;
  float xfactor=1.0/gs;
  float yfactor=1.0/gs;

  ac=bc=cc=dc=ec=fc=0;
  for(q=0; q<7; q++) cts[q]=0;
  for(i=0; i<XSIZ; i+=GRAN, xcount++, ycount=0) {
    for(j=0; j<YSIZ; j++) {
      for(k=0; k<32/GRAN; k++, ycount++) {
        for(ii=i; ii<i+GRAN  ii<XSIZ; ii++) {
          for(q=GRAN*k; q<GRAN*(k+1); q++) {
            cts[0]+=(a[ii*YSIZ+j]>>q)%2;
            cts[1]+=(b[ii*YSIZ+j]>>q)%2;
            cts[2]+=(c[ii*YSIZ+j]>>q)%2;
            cts[3]+=(d[ii*YSIZ+j]>>q)%2;
            cts[4]+=(e[ii*YSIZ+j]>>q)%2;
            cts[5]+=(f[ii*YSIZ+j]>>q)%2;
            cts[6]+=(r[ii*YSIZ+j]>>q)%2;
          }
        }
        result[ycount][xcount].Vx=SQRT3O2*(cts[0]+cts[1]-cts[3]-cts[4]);
        result[ycount][xcount].Vy=cts[2]-cts[5]+(.5*(cts[1]+cts[3]-cts[0]-cts[4]));
        total.Vx+=result[ycount][xcount].Vx;
        total.Vy+=result[ycount][xcount].Vy;
        result[ycount][xcount].Vx*=xfactor;
        result[ycount][xcount].Vy*=yfactor;
        ac+=cts[0];
        bc+=cts[1];
        cc+=cts[2];
        dc+=cts[3];
        ec+=cts[4];
        fc+=cts[5];
        statcount+=cts[6];
        for(q=0; q<7; q++) {  t1+=cts[q];  cts[q]=0;}
      }
    }
  }
  printf("total particles=%d\n", t1);
  printf("stationary particles=%d\n", statcount);
  printf("Vx=%f\nVy=%f\n", total.Vx/us, total.Vy/us);   /* These are the U values */
  printf("\n");
}
```

D.2 : En Cellang :

Cellang est un ensemble logiciel destiné à représenter, simuler et analyser une grande variété d'automates cellulaires. Il dispose de son propre langage, *Cellang*, de son compilateur, *cellc*, d'une machine virtuelle, *avcam*, et d'un programme de visualisation, *cellview*. Le tout est distribué gratuitement avec la licence GPL sur le site de son créateur, Dana Eckart :

http://www.cs.runet.edu/~dana/ca/cellular.html

type de structure : traitement pair/impair sur voisinage de Moore
collisions : apparemment FHPI, calcul (pas de LUT) pour le seul cas de collision frontale, extension simple du modèle HPP
performance : non testé, probablement autour de 1Mc/s

Malgré les faibles performances et le code rudimentaire, Cellang permet de "prototyper" des automates cellulaires de complexité arbitraire et d'en extraire les caractéristiques statistiques avant de commencer le codage de bas niveau. De plus, Cellang est un mélange de Fortran, de C et de Pascal qui n'est pas difficile à comprendre et à apprendre. Il reste tout de même incertain que *cellc* puisse générer du code très efficace (il semble générer du C) et donc un investissement très important doit être fourni pour améliorer la vitesse de simulation. Générer du code natif n'est qu'un premier pas, il faut aussi traduire efficacement l'abstraction de Cellang en un code qui soit en phase avec le microprocesseur. Comme en VHDL, le compilateur a plus de possibilités pour choisir les algorithmes et les structures de données, mais rien ne dit que le compilateur n'effectue pas une traduction simple, trandformant un *forall* en une véritable boucle (alors qu'en VHDL une boucle n'est pas traduite en un système itératif). Il reste donc beaucoup d'efforts à faire dans ce domaine mais cela ne pourra être entrepris que lorsque tous les mécanismes sous−jacents auront été étudié.

```
#  fhp
#  Copyright (C) 1994, 1995, 1997  J Dana Eckart
#
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 1, or (at your option)
#  any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with CELLULAR; see the file COPYING.  If not, write to the
#  Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

# An fhp latice gas automata with Cellang.
#
2 dimensions of
        const which of 0..1
        count of 0..6

        # The presence (1) of a particle traveling in the given direction.
        #
        particle[] for 6 of 0..1
end

# Hexagonal neighborhood.
#
neighbor[] for 6
        := [−1, 1], [0, 1], [1, 0], [0, −1], [−1, −1], [−1, 0] when cell.which
        := [0, 1], [1, 1], [1, 0], [1, −1], [0, −1], [−1, 0] otherwise
```

```
# Find out how many particles are incident on this cell.
#
count := 0
first := −1
forall i
        if neighbor[i].particle[i+%3] then
                count := count + 1
                if first < 0 then
                        first := i
                else
                        last  := i
                end
        end
end
cell.count := count

if count = 2 first+3 = last then
        if random%2 then
                # Rotate Clockwise
                #
                forall i
                        cell.particle[i+%1] := neighbor[i+%3].particle[i]
                end
        else
                # Rotate Counter−Clockwise
                #
                forall i
                        cell.particle[i−%1] := neighbor[i+%3].particle[i]
                end
        end
else
        # The incoming particles pass through unabated.
        #
        forall i
                cell.particle[i] := neighbor[i+%3].particle[i]
        end
end
```

D.3 : En FORTRAN :

En 1996, Oleh Baran (oleh@physics.mcgill.ca) m'a envoyé ce code en Fortran 77. Il ne m'a pas été utile car à l'époque je travaillais sur la technique de consultation de table et j'avais besoin d'une table, pas d'un calcul. De plus, les règles de collision sont FHPI, donc non saturées et sous−efficaces, sans même une particule immobile.

En revanche, il est intéressant de constater l'organisation du tableau principal, *NGAS*: c'est un tableau tridimensionnel, il suffit donc de changer l'ordre de déclaration des dimensions pour passer sans effort d'une organisation à une autre. Il ne faut pas oublier cependant que les ordres de déclaration sont différents en C et en Fortran. Enfin, la "consultation de table" dans ce code se résume à reconnaitre 5 cas (collisions frontales et triangulaires) et effectuer une rotation de l'octet si le cas est reconnu. Cette technique astucieuse n'est pas applicable à une échelle différente, par exemple en SIMD et/ou pour d'autres types de collisions.

type de structure : tableau multidimensionnel
collisions : FHPI, calcul (pas de LUT) des collisions frontales et triangulaires
performance : non mesurée. Selon l'auteur, l'efficacité est décevante, puisque les collisions sont réduites au minimum. Il n'a apparemment pas connaissance de méthodes plus efficace.

```
reproduit avec permission.

c********** one timestep in life of LGA *********
c************ by Oleh Baran ********************
c*********** Au 8th, 1994 ***********************

                SUBROUTINE timestep(CMP,NGAS)
                IMPLICIT REAL*4 (A−H,O−Z)
                COMMON /const/ LL1,LL2,DI
                complex CMP(0:LL1,0:LL2,1:2)
                integer NGAS(0:LL1+2,0:LL2+2,1:2)
                iidump=1

c−−−−−−−−−−−periodic boundary conditions−−−−−−−−−
            do 38 ivr = 1,2
            do 35 iv2 = 1,LL2+1
            NGAS(0,iv2,ivr) = NGAS(LL1+1,iv2,ivr)
            NGAS(LL1+2,iv2,ivr) = NGAS(1,iv2,ivr)
35          continue
            do 37 iv1 = 0,LL1+2
            NGAS(iv1,0,ivr) = NGAS(iv1,LL2+1,ivr)
            NGAS(iv1,LL2+2,ivr) = NGAS(iv1,1,ivr)
37          continue
38          continue
c−−−−−−−−−−−end of boundary conditions−−−−−−−−−−−

c−−−−−−−PROPAGATION−−−−−−−−−−−−−−−−−−−−−−−−−−−
                do 41 iv1 = 1,LL1+1
                do 43 iv2 = 1,LL2+1

                kr = NGAS(iv1,iv2+1,1)
                kk0 = and(1,kr)
                kr = NGAS(iv1−1,iv2,2)
                kk1 = and(2,kr)
                kr = NGAS(iv1−1,iv2−1,2)
                kk2 = and(4,kr)
                kr = NGAS(iv1,iv2−1,1)
                kk3 = and(8,kr) Au 8th, 1994
                kr = NGAS(iv1,iv2−1,2)
                kk4 = and(16,kr)
                kr = NGAS(iv1,iv2,2)
                kk5 = and(32,kr)
```

```
               rkk= real(kk0+kk1+kk2+kk3+kk4+kk5)
               CMP(iv1-1,iv2-1,1) = cmplx(rkk,0.0)
43             continue
41             continue
               do 45 iv1 = 1,LL1+1
                              do 47 iv2 = 1,LL2+1
                              kk0 = and(1,NGAS(iv1,iv2+1,2))
                              kk1 = and(2,NGAS(iv1,iv2+1,1))
                              kk2 = and(4,NGAS(iv1,iv2,1))
                              kk3 = and(8,NGAS(iv1,iv2-1,2))
                              kk4 = and(16,NGAS(iv1+1,iv2,1))
                              kk5 = and(32,NGAS(iv1+1,iv2+1,1))
                              rkk= real(kk0+kk1+kk2+kk3+kk4+kk5)
                              CMP(iv1-1,iv2-1,2) = cmplx(rkk,0.0)
47             continue
45             continue


c-------end of propagation-----------------------

c-------COLLISIONS--------------------------------
               do 77 iv = 1,2
               do 77 iv1 = 1,LL1+1
               do 77 iv2 = 1,LL2+1
               kk = int(real(CMP(iv1-1,iv2-1,iv)))
c - - - - - table of collisions - - - - - - - - - -

                       if (kk.EQ.18.OR.kk.EQ.36.OR.kk.EQ.9) THEN
                       kk = rotr(kk)
                       r3=ran3(iidump)
                       if (r3.LT.0.5) THEN
                       kk = rotr(kk)
                       end if

                       elseif (kk.EQ.42.OR.kk.EQ.21) THEN
                       kk = rotr(kk)
                       end if

c - - - - - end of table of collisions  - - - - -
               NGAS(iv1,iv2,iv) = kk
               bkk = real(bsum(kk)) - 6.*DI
c              bkk = real(bsum(kk))*(-1.)**(iv1+iv2) - 6.*DI
               CMP(iv1-1,iv2-1,iv) = cmplx(bkk,0.0)
77             continue
c-------end of collisions------------------------

               end
```

D.4 : En C :

LGAPACK est presque une implémentation de référence dans le domaine, il est écrit par Daniel H. Rothman (MIT) et Stéphane Zaleski (Paris 6).

C'est un ensemble logiciel destiné à l'étude des propriétés de différentes sortes de LGA, avec des versions à 6 et 7 vélocités, avec traitement par calcul booléen explicite ou non. Ils ont pour but de donner une idée de comment de tels codes peuvent être implémentés, pas d'être des outils réutilisables directement dans la pratique. Le tout est disponible sous licence GPL sur le serveur de l'université de Jussieu, ftp://ftp.jussieu.fr/jussieu/labos/lmm/Lgapack.

Ici nous nous intéressons à la version FHP saturée avec les plans de bits ("multisites").

type de structure : vectoriel, repère rhombohédrique (pas de pair/impair)
plans temporaires :Il semble que le code utilise deux tableaux séparés, un d'origine et un nouveau, et que les pointeurs soient échangés à chaque pas de temps.
collisions : FHP saturé
Dans la boucle principale, une variable aléatoire choisit entre une version pour chiralité droite et une version pour chiralité gauche.
performance : Non mesurée. Le but est surtout de mesurer les caractéristiques du fluide et de fonctionner sur des gros serveurs IBM ou SGI.

```
extrait de collision_7.c

/*
Lgapack Version VERSION for the simulation of flow with lattice-gas automata.
Copyright (C) 1997 D.H. Rothman and S. Zaleski.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

dan@segovia.mit.edu, zaleski@lmm.jussieu.fr

*/

void collision_7_right(unsigned int a, unsigned int b,unsigned int c,unsigned int d,
     unsigned int e, unsigned int f, unsigned int *g, unsigned int *na, unsigned int *nb,
     unsigned int *nc,unsigned int *nd, unsigned int *ne, unsigned int *nf,unsigned int s)
{
   unsigned int t1,t2,t3,t4,t5,t6,u1,u2,u3,g1,g2,g3,del,c0,c1,c2,c3,c4,c5,c6,e1,e2,e3,e4,e5,e6;

/* YG :
 rappel sur les directions :
     C      B
      \    /
       \  /
  D---- G ----A
       / \
      /   \
     E     F
*/

   t1 = a^b;
   t2 = b^c;
   t3 = c^d;
   t4 = d^e;
   t5 = e^f;
   t6 = f^a;
```

```
   e1 = t1 t6 ( *g^b );
   e2 = t2 t1 ( *g^c );
   e3 = t3 t2 ( *g^d );
   e4 = t4 t3 ( *g^e );
   e5 = t5 t4 ( *g^f );
   e6 = t6 t5 ( *g^a );

   u1 = t1 t4;
   u2 = t2 t5;
   u3 = t3 t6;

/* g means pair of opposing particles */
   g1 = u1 ( b^d );
   g2 = u2 ( c^e );
   g3 = u3 ( d^f );

/* del means triple collision, with or without rest particles */
   del = u1 u2 u3;

/* c0 is condition for involving a rest particle */

/* YG : version originale : (mauvaise recopie de d'Humières)
   c0 = ~(del | ( g1 | (e1^(~e4))) ( g2 | (e2 ^ (~e5))) ( g3 | (e3^(~e6))));
*/
   c0 = ~(del | (( g1 | (e2^(~e5))) ( g2 | (e3 ^ (~e6))) ( g3 | (e1^(~e4)))));
/* YG: correctif (non vérifié) d'Andreas Buness, Osnabrück, abuness@uos.de */

   /* rotate right */
   /* If no c), as in 6 particle. Else, collide with rest part. */

   c1 = del | ((~c0) (g3 | g1 ~g2)) | (c0 (e2 | ( e1 (~e5)) | e6 ( ~e4) ));
   c2 = del | ((~c0) (g1 | g2 ~g3)) | (c0 (e3 | ( e2 (~e6)) | e1 ( ~e5) ));
   c3 = del | ((~c0) (g2 | g3 ~g1)) | (c0 (e4 | ( e3 (~e1)) | e2 ( ~e6) ));
   c4 = del | ((~c0) (g3 | g1 ~g2)) | (c0 (e5 | ( e4 (~e2)) | e3 ( ~e1) ));
   c5 = del | ((~c0) (g1 | g2 ~g3)) | (c0 (e6 | ( e5 (~e3)) | e4 ( ~e2) ));
   c6 = del | ((~c0) (g2 | g3 ~g1)) | (c0 (e1 | ( e6 (~e4)) | e5 ( ~e3) ));

#ifdef SOLID_SITES
   *na =((~s) (c1^a)) |( s d);
   *nb =((~s) (c2^b)) |( s e);
   *nc =((~s) (c3^c)) |( s f);
   *nd =((~s) (c4^d)) |( s a);
   *ne =((~s) (c5^e)) |( s b);
   *nf =((~s) (c6^f)) |( s c);
   *g =((~s) (c0^*g)) |( s *g);
#else
   *na = c1^a;
   *nb = c2^b;
   *nc = c3^c;
   *nd = c4^d;
   *ne = c5^e;
   *nf = c6^f;
   *g = c0^*g;
#endif
}
```

D.5 : Toujours en C :

Ce code est curieux pour plusieurs raisons, surtout par la manière dont il traite la structure des données, par exemple en utilisant une syntaxe du langage C qui permet une utilisation indépendante de la représentation en mémoire. Il reste à vérifier l'impact de ce choix sur les performances (mesurer avec une configuration et une autre). Ce code montre qu'on peut programmer quelque chose de classique tout en étant étrange.

type de structure : selon #define, mais orienté octets, avec lignes paires et impaires.
collisions : FHPI (5 cas), 6 bits
buffer temporaire : 2 plans
performance : Non mesurée.

```
/*  fhplattice.c

    downloaded from
    http://www.media.mit.edu/people/wad/mas864/src/fhplattice.txt

    This application solves for the behavior of a lattice gas,
    using cellular automata in a hexagonal lattice (FHP)

    J. Watlington, 10/25/95

    Usage:  fhplattice [ block | random | test ] [ -s <array_size> ]

    Modified:
*/

#include <stdio.h>
#include <stdlib.h>
#include "display.h" /* YG : non inclus */

#define SCATTER_ALTERNATELY
#define DEFAULT_MAX_ITERATIONS  2000
#define MAX_ARRAY_SIZE          1024
#define DEFAULT_ARRAY_SIZE      512

/*  The following define how the array is seeded  */

#define SEED_BLOCK_SIZE         ((array_size) / 4)
#define SEED_BLOCK_START        ((3 *(array_size)) / 8)
#define SEED_RAND_SEED          0xFF67533
#define SEED_RAND_THRESHOLD     22000
#define SEED_BLOCK              0
#define SEED_BLOCK_AND_RND      1
#define SEED_TEST               2
#define DEFAULT_ARRAY_SEED      SEED_BLOCK_AND_RND

/*  The following global variables are used for command line parameters */

int   array_size = DEFAULT_ARRAY_SIZE;
int   array_seed = DEFAULT_ARRAY_SEED;
int   max_iterations = DEFAULT_MAX_ITERATIONS;

/*  This is the structure that will represent the lattice of nodes  */

#define DEGREE_OF_INTERCONNECT   6
#define NUM_RULES                64   /* 2^DEGREE_OF_INTERCONNECT */
#define MAX_PARTICLES            6    /* used only for display  */

#ifndef USE_BIT_STORAGE
typedef struct {
  char    northeast;
```

```
  char    northwest;
  char    east;
  char    west;
  char    southeast;
  char    southwest;
} lattice_node;
#else
typedef struct {
  char    northeast:1,
          northwest:1,
          east:1,
          west:1,
          southeast:1,
          southwest:1,
          reserved:2;
} lattice_node;
#endif

/*  One of the few peculiarities of this program is the manner in
    which I represent the lattice arrays.  For ease of access, I
    store an array of pointers (d) to the rows of the array.   */

typedef struct {
  int           size;     /*  Assume square array !  */
  lattice_node  *d[ MAX_ARRAY_SIZE ];
} lattice;

/*  These are the rules the determine the behavior of this lattice gas */

lattice_node  rules[ NUM_RULES ] = {  /* { NE, NW, E, W, SE, SW }   Col. Scatter */
  { 0,0,0,0,0,0 }, { 1,0,0,0,0,0 }, { 0,1,0,0,0,0 }, { 1,1,0,0,0,0 },
  { 0,0,1,0,0,0 }, { 1,0,1,0,0,0 }, { 0,1,1,0,0,0 }, { 1,1,1,0,0,0 },
  { 0,0,0,1,0,0 }, { 1,0,0,1,0,0 }, { 0,1,0,1,0,0 }, { 1,1,0,1,0,0 },
  { 0,1,0,0,1,0 }, { 1,0,1,1,0,0 }, { 0,1,1,1,0,0 }, { 1,1,1,1,0,0 },  /* 0 2b */

  { 0,0,0,0,1,0 }, { 1,0,0,0,1,0 }, { 0,0,1,1,0,0 }, { 1,1,0,0,1,0 },  /* 2 2b */
  { 0,0,1,0,1,0 }, { 1,0,1,0,1,0 }, { 0,1,1,0,1,0 }, { 1,1,1,0,1,0 },
  { 0,0,0,1,1,0 }, { 0,1,1,0,0,1 }, { 0,1,0,1,1,0 }, { 1,1,0,1,1,0 },  /* 1 3b */
  { 0,0,1,1,1,0 }, { 1,0,1,1,1,0 }, { 0,1,1,1,1,0 }, { 1,1,1,1,1,0 },

  { 0,0,0,0,0,1 }, { 0,0,1,1,0,0 }, { 0,1,0,0,0,1 }, { 1,1,0,0,0,1 },  /* 1 2b */
  { 0,0,1,0,0,1 }, { 1,0,1,0,0,1 }, { 1,0,0,1,1,0 }, { 1,1,1,0,0,1 },  /* 2 3b */
  { 0,0,0,1,0,1 }, { 1,0,0,1,0,1 }, { 0,1,0,1,0,1 }, { 1,1,0,1,0,1 },
  { 0,0,1,1,0,1 }, { 1,0,1,1,0,1 }, { 0,1,1,1,0,1 }, { 1,1,1,1,0,1 },

  { 0,0,0,0,1,1 }, { 1,0,0,0,1,1 }, { 0,1,0,0,1,1 }, { 1,1,0,0,1,1 },
  { 0,0,1,0,1,1 }, { 1,0,1,0,1,1 }, { 0,1,1,0,1,1 }, { 1,1,1,0,1,1 },
  { 0,0,0,1,1,1 }, { 1,0,0,1,1,1 }, { 0,1,0,1,1,1 }, { 1,1,0,1,1,1 },
  { 0,0,1,1,1,1 }, { 1,0,1,1,1,1 }, { 0,1,1,1,1,1 }, { 1,1,1,1,1,1 },
};

#ifdef SCATTER_ALTERNATELY
static int           scatter_dir = 0;
static lattice_node  scatter[ 4 ] = {
  { 0,0,1,1,0,0 }, { 0,1,0,0,1,0 }, { 1,0,0,0,0,1 }, { 0,0,1,1,0,0 }
};
#endif

/*  These are prototypes of functions internal to this code module */

void parse_arguments( int argc, char **argv );
lattice *alloc_lattice( int size );
void seed_lattice( lattice *l, int seed );
void calc_collisions( lattice *l, lattice *temp );
void calc_collision_row( lattice *in, lattice *out,
   int prev_row_index, int cur_row_index, int next_row_index );
void display_lattice( window_tag wt, lattice *l );

/*  This is the top level of the program  */

void main( int argc, char **argv )
{
  lattice    *nodes[ 2 ];
  int        next_buf = 1;
```

```
  int        cur_buf = 0;
  int        num_iters = 0;
  window_tag w;

  parse_arguments( argc, argv );
  nodes[ cur_buf ] = alloc_lattice( array_size );
  nodes[ next_buf ] = alloc_lattice( array_size );
  seed_lattice( nodes[ cur_buf ], array_seed );

  if( (w = display_init_window( "FHP", array_size, array_size,
       MAX_PARTICLES + 1)) < 0 )
  {
    printf( "Unable to open display window.\n" );
    exit( -1 );
  }
  display_lattice( w, nodes[ cur_buf ] );

  while( num_iters++ < max_iterations ) {
    calc_collisions( nodes[ cur_buf ], nodes[ next_buf ] );
    display_lattice( w, nodes[ next_buf ] );

    cur_buf = next_buf;
    next_buf = 1 - next_buf;
  }
}

void calc_collisions( lattice *in, lattice *out )
{
  int           size = in->size - 1;
  int           row;

  calc_collision_row( in, out, size, 0, 1 );

  for( row = 1; row < size; row++ )
    calc_collision_row( in, out, row - 1, row, row + 1 );

  calc_collision_row( in, out, size - 1, size, 0 );
}

void calc_collision_row( lattice *in, lattice *out,
   int prev_row_index, int cur_row_index, int next_row_index )
{
  int      adr;
  int      col;
  int      size = in->size - 1;

  lattice_node   *outcome;
  lattice_node   *cur_row_in = in->d[ cur_row_index ];
  lattice_node   *prev_row_out = out->d[ prev_row_index ];
  lattice_node   *cur_row_out = out->d[ cur_row_index ];
  lattice_node   *next_row_out = out->d[ next_row_index ];

  /*  Special case the first node in a row  */

  adr = 0;
  if( cur_row_in->northeast != 0 )  adr |= 32;
  if( cur_row_in->northwest != 0 )  adr |= 16;
  if( cur_row_in->east  != 0 )  adr |= 8;
  if( cur_row_in->west  != 0 )  adr |= 4;
  if( cur_row_in->southeast != 0 )  adr |= 2;
  if( cur_row_in->southwest != 0 )  adr |= 1;

  outcome = rules + adr;

  if( cur_row_index 1 )
  {  /* odd row */
    prev_row_out[ 0 ].southwest = outcome->northeast;
    prev_row_out[ size ].southeast = outcome->northwest;  /* boundary */
    cur_row_out[ 1 ].west = outcome->east;
    cur_row_out[ size ].east  = outcome->west;   /* periodic boundary ! */
    next_row_out[ 0 ].northwest = outcome->southeast;
    next_row_out[ size ].northeast = outcome->southwest;  /* boundary */
  }
  else
```

```
  {  /* even row − offset in slightly */
    prev_row_out[ 1 ].southwest = outcome−>northeast;
    prev_row_out[ 0 ].southeast = outcome−>northwest;
    cur_row_out[ 1 ].west = outcome−>east;
    cur_row_out[ size ].east  = outcome−>west;   /* periodic boundary ! */
    next_row_out[ 1 ].northwest = outcome−>southeast;
    next_row_out[ 0 ].northeast = outcome−>southwest;
  }

  /*  Now process the inner nodes  */

  for( col = 1; col < size; col++ ) {
    cur_row_in++;
    adr = 0;
    if( cur_row_in−>northeast != 0 )  adr |= 32;
    if( cur_row_in−>northwest != 0 )  adr |= 16;
    if( cur_row_in−>east  != 0 )  adr |= 8;
    if( cur_row_in−>west  != 0 )  adr |= 4;
    if( cur_row_in−>southeast != 0 )  adr |= 2;
    if( cur_row_in−>southwest != 0 )  adr |= 1;

    outcome = rules + adr;

#ifdef SCATTER_ALTERNATELY
    switch( adr ) {
      case 0x21:
        outcome =  scatter_dir ];
        scatter_dir = 1 − scatter_dir;
        break;

      case 0x0C:
        outcome =  scatter_dir + 1 ];
        scatter_dir = 1 − scatter_dir;
        break;

      case 0x12:
        outcome =  scatter_dir + 2 ];
        scatter_dir = 1 − scatter_dir;
        break;
      }
#endif

    if( cur_row_index 1 )
    {  /* odd row */
      prev_row_out[ col ].southwest = outcome−>northeast;
      prev_row_out[ col − 1 ].southeast = outcome−>northwest;
      cur_row_out[ col + 1 ].west = outcome−>east;
      cur_row_out[ col − 1 ].east  = outcome−>west;
      next_row_out[ col ].northwest = outcome−>southeast;
      next_row_out[ col − 1 ].northeast = outcome−>southwest;
    }
    else
    {  /* even row − offset in slightly */
      prev_row_out[ col + 1 ].southwest = outcome−>northeast;
      prev_row_out[ col ].southeast = outcome−>northwest;
      cur_row_out[ col + 1 ].west = outcome−>east;
      cur_row_out[ col − 1 ].east  = outcome−>west;
      next_row_out[ col + 1 ].northwest = outcome−>southeast;
      next_row_out[ col ].northeast = outcome−>southwest;
    }
  }

  /* Special case the last node in a row  */

  cur_row_in++;
  adr = 0;
  if( cur_row_in−>northeast != 0 )  adr |= 32;
  if( cur_row_in−>northwest != 0 )  adr |= 16;
  if( cur_row_in−>east  != 0 )  adr |= 8;
  if( cur_row_in−>west  != 0 )  adr |= 4;
  if( cur_row_in−>southeast != 0 )  adr |= 2;
  if( cur_row_in−>southwest != 0 )  adr |= 1;

  outcome = rules + adr;
```

```
  if( cur_row_index 1 )
  {  /* odd row */
    prev_row_out[ size ].southwest = outcome−>northeast;
    prev_row_out[ size − 1 ].southeast = outcome−>northwest;
    cur_row_out[ 0 ].west = outcome−>east;    /* periodic boundary ! */
    cur_row_out[ size − 1 ].east  = outcome−>west;
    next_row_out[ size ].northwest = outcome−>southeast;
    next_row_out[ size − 1 ].northeast = outcome−>southwest;
  }
  else
  {  /* even row − offset in slightly */
    prev_row_out[ 0 ].southwest = outcome−>northeast;  /* boundary */
    prev_row_out[ size ].southeast = outcome−>northwest;
    cur_row_out[ 0 ].west = outcome−>east;   /* periodic boundary ! */
    cur_row_out[ size − 1 ].east  = outcome−>west;
    next_row_out[ 0 ].northwest = outcome−>southeast;  /* boundary */
    next_row_out[ size ].northeast = outcome−>southwest;
  }
}

void parse_arguments( int argc, char **argv )
{
  int   index;

  if( argc > 1 )
  {
    index = 1;
        while( index < argc )
        {
          if( argv[ index ][0] == '−' )
          {   /*  − option, only −size defined right now  */
            if( index + 1 >= argc )
                {
                  printf( "%s option requires an argument !\n", argv[ index ] );
                  exit( −1 );
                }
                array_size = atoi( argv[ ++index ] );
                printf( "Simulating a %d x %d array\n", array_size, array_size );
          }
          else
          {   /*  must be array seed − block, rand, or test  */
        switch( argv[ index ][0] ) {
                case 'b':
                        array_seed = SEED_BLOCK;
                        break;
                case 'r':
                        array_seed = SEED_BLOCK_AND_RND;
                        break;
                case 't':
                    array_seed = SEED_TEST;
                        break;
                default:
                  printf( "I don't understand %s as a seed pattern !\n",
                    argv[ index ] );
                  exit( −1 );
                }
          }
          index++;
        }
  }
  srand( SEED_RAND_SEED );
}

void seed_lattice( lattice *l, int seed )
{
  int   row, col;
  int   lattice_size = l−>size;
  lattice_node  *n;

  /* Now seed the lattice structure.  It has already
     been cleared to all zeroes.                    */

  if( seed == SEED_BLOCK_AND_RND )
```

```
    for( row = 0; row < lattice_size; row++ ) {
      for( n = l->d[ row ], col = 0; col < lattice_size; col++, n++ ) {
        if( rand() > SEED_RAND_THRESHOLD) n->northeast = 1;
        if((rand() > SEED_RAND_THRESHOLD) n->northwest = 1;
        if((rand() > SEED_RAND_THRESHOLD) n->east = 1;
        if((rand() > SEED_RAND_THRESHOLD) n->west = 1;
        if((rand() > SEED_RAND_THRESHOLD) n->southeast = 1;
        if((rand() > SEED_RAND_THRESHOLD) n->southwest = 1;
      }
    }

  if( (seed == SEED_BLOCK) || (seed == SEED_BLOCK_AND_RND) )
    for( row = SEED_BLOCK_START;
         row < (SEED_BLOCK_START + SEED_BLOCK_SIZE); row++ ) {
      n = l->d[ row ] + SEED_BLOCK_START;
      for( col = SEED_BLOCK_START;
          col < (SEED_BLOCK_START + SEED_BLOCK_SIZE); col++, n++ ) {
        n->northeast = 1;
        n->northwest = 1;
        n->east  = 1;
        n->west  = 1;
        n->southeast = 1;
        n->southwest = 1;
      }
    }

  if( seed == SEED_TEST ) {
    for( n = l->d[ 0 ], col = 0; col < l->size; col++, n++ ) {
      n->east  = 1;
      n->southwest = 1;
    }

    for( row = 2; row < (array_size - 1); row++ ) {
      n = l->d[ row ] + 4 + (row / 2);
      n->northeast = 1;
      n->west  = 1;
    }
  }
}

/*  alloc_lattice
    This lattice is stored as an array of lattice_nodes, where each
    row is pointed to independently (although in this case the lines
    are actually stored consecutively.)
*/
lattice *alloc_lattice( int lattice_size )
{
  int           row, col;
  long          total_array_size;
  lattice_node  *n;
  lattice_node  *row_start;
  lattice       *l;

  /* malloc the memory */

  total_array_size = (lattice_size * lattice_size * sizeof(lattice_node))
    + sizeof( lattice );

  if( (l = (lattice *)malloc( total_array_size )) == NULL ) {
    printf( "unable to alloc %d bytes of memory for lattice\n",
        total_array_size );
    exit( -1 );
  }

  /* Now init the lattice structure.  */

  l->size = lattice_size;
  row_start = (lattice_node *)((long)l + sizeof( lattice ));
  for( row = 0; row < lattice_size; row++ ) {
    l->d[ row ] = row_start;

    for( n = row_start, col = 0; col < lattice_size; col++, n++ ) {
      n->northeast = 0;
      n->northwest = 0;
```

```
      n->east = 0;
      n->west = 0;
      n->southeast = 0;
      n->southwest = 0;
    }
    row_start += lattice_size;
  }
  return( l );
}

void display_lattice( window_tag wt, lattice *l )
{
  int           x, y;
  int           size = l->size;
  int           num_particles;
  lattice_node  *n;

  for( y = 0; y < size; y++ )
  {
    n = l->d[ y ];
    for( x = 0; x < size; x++, n++ )
    {
      num_particles = n->northeast;
      num_particles += n->northwest;
      num_particles += n->east;
      num_particles += n->west;
      num_particles += n->southeast;
      num_particles += n->southwest;
      display_set_pixel( wt, x, y, num_particles );
    }
  }
  display_update( wt );
}
```

D.6 : Dans un hypercube iPSC :

En 1987, Fung F. Lee (Stanford University) et George B. Adams (Purdue) ont conduit des recherches (NCC 2−387) sur la parallélisation des codes style FHP dans un hypercube à 32 processeurs au NASA Ames Research Center. Ils observent dans leur rapport que le *speedup* est quasi linéaire (x28 / 32 CPU). Ils expliquent aussi leur amélioration de la technique de passage de messages (*piggy−back*) : comme un seul bit doit être envoyé vers le nord−est (ou sud−est) le message est envoyé en deux *hops*, grâce à un ordonnancement des envois et un tag sur les messages.

Le modèle à 6 bits est traité par calcul booléen. Ici, les deux phases, translation et collision, sont indépendents pour des raisons de communication intra−CPU. Le mélange entre les deux phase est plus difficile qu'avec un ordinateur à mémoire partagée et mérite qu'on s'y attarde, mais il dépend étroitement de l'architecture cible. Le problème de la complexité du calcul booléen est aussi abordé mais la méthode de résolution n'est pas exposée. Il semble qu'ils aient employé une méthode "directe", descriptive, vu la simplicité des règles de collision (comme en FHP1 mais plus complexe).

type de structure : distribuée, 7 plans de bits par noeud, lignes paires/impaires (ce qui pose un problème de topologie dans un hypercube...)
collisions : FHPII 6 bits *(note : mêmes règles que celles utilisées par S. Wolfram sur sa CM1)*
buffer temporaire :
performance : 50Kc/s (donc 1,4Mc/s avec 32 noeuds)

```
#define WSIZE   32
#define MSB     31
typedef unsigned long Word;

/* There are YN by XN*WSIZE bits in each of the state−bit planes */
/* YN must be even */
Word s0[YN][XN],s1[YN][XN],s2[YN][XN],s3[YN][XN],s4[YN][XN],s5[YN][XN];
Word obstacle[YN][XN];

/* This is the main computation loop which updates the cells. */
void fluid_flow()
{
        for (t = 0; t < TMAX; t++) {
                advance();
                collide();
        }
}

/* YG : translation : */

#define AllY  for (y=0; y<YN; y++)
#define EvenY for (y=0; y<YN; y+=2)
#define OddY  for (y=1; y<YN; y+=2)
#define AllX  for (x=0; x<XN; x++)

#define RowLS(s) for (x=0; x<XN−1; x++) s[y][x] = s[y][x]<<1 | s[y][x+1]>>MSB; \
        s[y][XN−1]<<=1;

#define RowRS(s) for (x=XN−1; x>0; x−−) s[y][x] = s[y][x]>>1 | s[y][x−1]<<MSB; \
        s[y][0]>>=1;

#define ColDS(s) for (y=0; y<YN−1; y++) s[y][x] = s[y+1][x];

#define ColUS(s) for (y=YN−1; y>0; y−−) s[y][x] = s[y−1][x];

/* advance() moves state bits by working on the bit planes one at a
|| time. This is a over−simplified version ignoring boundary cells and
|| communication.
*/
```

```
void advance()
{
        int x, y;

        AllY  {RowRS(s0)}
        EvenY {RowRS(s1)}       AllX {ColUS(s1)}
        OddY  {RowLS(s2)}       AllX {ColUS(s2)}
        AllY  {RowLS(s3)}
        OddY  {RowLS(s4)}       AllX {ColDS(s4)}
        EvenY {RowRS(s5)}       AllX {ColDS(s5)}
}

/* Algorithm for advance with piggyback communication :
|| This is a simplified version of advance() with pseudo code.
|| Conceptually, the following action happens for each of the six
|| non−obstacle bit planes.
|| The boundary bits on a specific side of a bit plane are gathered
|| into a message buffer, and sent to a specific neighbor.
|| The message is then received by the neighbor, and the bits of the
|| message are scattered to a specific boundary of the same bit plane
|| of the neighbor.
*/
void advance()
{
        int x, y;

        /* R−bits, L−bits, U−bits and D−bits are the bits on the
        || right,left, up (top) and down (bottom) boundaries respectively.
        */

        {Gather R−bits from s0,s1,s5 into a message, and Send to RIGHT.}
        AllY  {RowRS(s0)}
        EvenY {RowRS(s1)}
        EvenY {RowRS(s5)}
        {Receive a message from LEFT, and scatter bits to s0,s1,s5.}

        {Gather L−bits from s3,s4,s2 into a message, and Send to LEFT.}
        AllY  {RowLS(s3)}
        OddY  {RowLS(s4)}
        OddY  {RowLS(s2)}
        {Receive a message from RIGHT, and scatter bits to s3,s4,s2.}

        {Gather U−bits from s1,s2 into a message, and Send to UP.}
        AllX  {ColUS(s1)}
        AllX  {ColUS(s2)}
        {Receive a message from DOWN, and scatter bits to s1,s2.}

        {Gather D−bits from s4,s5 into a message, and Send to DOWN.}
        AllX  {ColDS(s4)}
        AllX  {ColDS(s5)}
        {Receive a message from UP, and scatter bits to s4,s5.}
}

/* collide() effectively transforms the old states into new states
|| for WSIZE (32) cells at a time in its inner most loop by evaluating
|| the boolean expressions which encode the set of interaction rules.
*/
void collide()
{
        int x, y;
        Word obs, a, b, c, d, e, f;
        Word Ca, Cb, Cc, abcdef;

        for (y = 0; y < YN; y++) {
                for (x = 0; x < XN; x++) {
                        obs = obstacle[y][x];
                        a = s0[y][x]; b = s1[y][x]; c = s2[y][x];
                        d = s3[y][x]; e = s4[y][x]; f = s5[y][x];

                        abcdef = a ~b c ~d e ~f |
                                ~a b ~c d ~e f;

                        Ca = ~(a|d) b e | a d ((cf ~(b|e) |
                                        ~(c|f) ~(be) | abcdef;
```

```
                        Cb = ~(b|e) c f | b e ((da ~(c|f) |
                                        ~(d|a) ~(cf) | abcdef;
                        Cc = ~(c|f) d a | c f ((eb ~(d|a) |
                                        ~(e|b) ~(da) | abcdef;

                        s0[y][x] = ~obs (Ca ^ a) | obs d;
                        s1[y][x] = ~obs (Cb ^ b) | obs e;
                        s2[y][x] = ~obs (Cc ^ c) | obs f;
                        s3[y][x] = ~obs (Ca ^ d) | obs a;
                        s4[y][x] = ~obs (Cb ^ e) | obs b;
                        s5[y][x] = ~obs (Cc ^ f) | obs c;
                }
        }
}
```

D.7 : Conclusion :

Les codes pour calculer FHP sont très variés et il serait difficile d'en faire une analyse comparée poussée. Chacun peut s'inspirer des exemples de cette annexe pour concevoir et explorer l'algorithme qui lui plaît, en gardant à l'esprit que "Le code le plus rapide n'existe pas" (M. Abrash).

Toutefois, la lecture de ces exemples permet de dégager quelques axes et règles de conception pour les programmes FHP ou similaires :

- Utiliser les règles de codage saturées (au moins) pour que le calcul soit efficace. L'exemple d'Oleh Baran montre que des règles incomplètes font travailler l'ordinateur vite mais pour presque rien, puisque le *mean free path* est très long. Les collisions saturées sont plus compliquées mais il faut moins de sites et de pas de calcul pour arriver au résultat. Le caractère *memory bound* du modèle fait pencher la balance encore plus vers FHP3+.

- Veiller à ce que le code reste indépendent d'une géométrie ou d'une taille de mots particulière. Par exemple en 1), le tunnel est défini par plusieurs #define (ce que l'auteur a modifié plus tard), et la plupart des codes utilisent des tailles de mots de 32 bits. Bien que cela soit portable (puisque c'est le souci de nombreuses personnes), il reste encore des plateformes 16 bits et les plateformes 64 et 128 bits se démocratisent. Il faut donc pouvoir définir la taille des mots et des géométries à un seul endroit. Comme cela a été fait dans le programme du mémoire, il est recommandé de rendre la taille du tunnel variable, afin de pouvoir "accélérer" les calculs en redimensionnant le tunnel en cours de route.

- Les techniques de strip−mining et d'affichage progressif n'ont pas été utilisées : il est recommandé d'en tenir compte dès le début de la conception du programme car elles l'influencent profondément. Les programmes présentés ne visent pas des performances exceptionnelles et ont été écrits pour des stations de travail, ce qui explique que le strip−mining n'ait pas été étudié. Cependant, l'affichage des particules, pas à pas, permet de vérifier formellement le comportement du fluide comme des règles de collision. Une interface graphique, même rudimentaire, aurait permis de trouver rapidement l'erreur commise dans LGAPACK. Une erreur minime arrive tellement vite qu'il est parfois trop tard pour réparer l'erreur, et l'interface graphique du projet est un élément essentiel pour le développement et le débogage du programme, en absence de débogueur.

- Quelle méthode de déplacement choisir ? Celle qui effectue le moins d'opérations possible et qui favorise la localité des données. En particulier, il faut réduire le nombre d'accès désordonnés et redondants à la mémoire, en faisant attention à la taille des lignes de caches. Ainsi, sur une station de travail classique, il faut mélanger la phase de calcul et de déplacement car les données se trouvent au même endroit au même moment. Or, dans la plupart des cas rencontrés, les phases sont séparées : tout le tableau est calculé, puis le déplacement est effectué (voir 1) et LGAPACK). Cette approche ralentit le programme considérablement car il y a deux fois plus de *cache miss*. En concentrant le calcul et le déplacement, on peut en profiter pour traiter et afficher les données.

- Il est inutile, pour un programme qui veut être rapide, de recourir à de nombreux appels de sous−fonctions : dans l'exemple 1) (*void nextstate()*) et dans LGAPACK, la boucle centrale contient de nombreux appels de routines qui bénéficieraient de l'*inlining*. Dans LGAPACK, la fonction de calcul d'un site est appelée *pour chaque site*, inutilement : il serait possible avec un simple

copier/coller de gagner quelques pourcents sur la durée d'exécution du programme...

- L'orientation et la projection du réseau hexagonal sur le réseau carré sont des choix qu'il faut mûrir patiemment. Dans le corps du mémoire, nous avons vu que les lignes paires et impaires ont d'abord été orientées horizontalement, puis verticalement pour le modèle à 32 bits et enfin horizontalement pour le travail du mémoire, à cause de propriétés différentes du traitement. Le réseau rhombohédrique de LGAPACK est une alternative intéressante mais son impact sur les algorithmes annexes est encore inconnu.

- La chiralité est un sujet délicat :
  − certains codes choisissent la chiralité au hasard au début du pas de temps (LGAPACK). Cela peut rendre le développement, les vérifications et le débogage difficiles. La simplicité est assurée mais l'impact sur la chiralité générale du modèle n'a pas été étudiée à ma connaissance.
  − D'autres choisissent la chiralité en fonction de la parité du numéro de la ligne (H. Stockman entre autres). Cette technique est déconseillée.
  − Enfin la technique choisie dans le mémoire est d'utiliser le "bruit brownien" du domaine d'étude pour choisir, site par site, la chiralité. Le compromis est intéressant car la chiralité n'est pas corrélée avec la présence ou l'absence des particules. Dans le cas où il faudrait gagner encore quelques cycles lors du calcul, la première technique (LGAPACK) serait utilisée mais la taille du code de détection risque de doubler et de ralentir le développement du programme.