
# Systèmes Distribués
# Rapport de projet

# Distributed Bag of Tasks

Ferhat Yousfi, Jeremy Ballot

December 20, 2014

# Sommaire

# 1 Introduction

Les travaux exposés dans ce rapport ont pour but de nous faire découvrir un nouveau type de programmation qui est la programmation distribuée.
Ce projet consiste à implémenter le modèle *Distributed Bag of Tasks* sur le célèbre "Jeu de la Vie".

# 2 Distributed Bag of Tasks

Le modèle *Distributed Bag of Tasks* permet l'exécution de plusieurs tâches de façon distribuée, il se repose sur une architecture *Serveur - Clients*.
Son principe est de définir:

- Un coordinateur:

  C'est le centre du programme, il définit la liste des tâches a effectuer, il distribue les tâches aux assistants, reçoit le résultat de chaque tâche.
  Il coordonne les assistants qui se connectent, communique avec eux en suivant un schéma de communication définit, et synchronise les exécutions des tâches et leurs envois.

- Les assistants:

  Ce sont des programmes qui s'exécutent sur une ou plusieurs machines distantes et qui communiquent avec le coordinateur (serveur), leur rôle est de traiter plusieurs tâches (envoyées par le coordinateur) et de transmettre les résultats, le tout en suivant le même schéma de communication que celui du coordinateur.

- Schéma de communication:

  Afin d'assurer une bonne coordination, il est impératif de communiquer, et que chaque partie comprenne ce que l'autre dit. Dans ce cas définir un schéma de communication s'avère trés important, ce dernier doit comporter plusieurs types de message et pour chaque message reçu ou envoyé, une opération s'effectue.

Maintenant que nous avons definit d'une façon globale le principe du *Distributed Bag of Tasks*, nous allons passer à l'explication de notre projet.

# 3 Travaux réalisés

Nous allons maintenant expliquer les principaux objectifs que nous avons réalisé ainsi que leur implémentation, il nous a été demandé d'utiliser le langage *C* pour coder notre projet.

## 3.1 Serveur-Clients

Nous allons voir l'implémentation du serveur.

### 3.1.1 Le Serveur

Nous avons mis en place un serveur utilisant un socket où un seul client peut se connecter, afin de permettre la connexion de plusieurs clients sur le même serveur, nous avons apporté les modification suivantes:

- Tableau de socket:

  Nous avons ajouter une structure Client qui contient un socket qui va connecter un client, puis un tableau qui contient des clients.

```
  /*init connexion */
  int init_connect(){

SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    SOCKADDR_IN sin = { 0 };
SOCKADDR_IN csin;
socklen_t client_len=sizeof(csin);
if(sock < 0)
    {
perror("Erreur de socket");
}
    int optval=200;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,&optval, sizeof (optval));

sin.sin_addr.s_addr =htonl(INADDR_ANY);/*inet_addr("192.168.1.67");*/
    sin.sin_port = htons(11550);
    sin.sin_family = AF_INET;
if(bind(sock,(SOCKADDR *) &sin, sizeof sin) == -1)
    {
       perror("Erreur de bind");
       }
if(listen(sock,18) == -1)
{
perror("Erreur de réception");

}
  return sock;
}

typedef struct
  {
    SOCKET sock;
  } Client;
```

- Connecteur de Clients:

  À l'aide d'une fonction, connecteur_client, qui écoute le socket du serveur constamment, chaque tentative de connexion d'un client est repérée et acceptée. Cette acceptation déclenche l'enregistrement du descripteur de socket dans une case du tableau de socket avant de repasser à l'écoute de la connexion suivante.
  Ainsi plusieurs clients peuvent se connecter sur notre serveur.
  Au cours du traitement, si un client se déconnecte, son socket est automatiquement supprimé du tableau.

```
  void connecteur_client ( Client *client_tab, int *nbr_client){
SOCKADDR_IN sin = { 0 };
SOCKADDR_IN csin;
socklen_t client_len=sizeof(csin);
SOCKET sock=init_connect();

     do {
  client_tab[nbr_client[0]].sock=accept(sock,(struct sockaddr *)&csin,&client_len);
  printf("Un client vient se connecte avec la socket %d\n",nbr_client[0]);
nbr_client[0]++;} while(nbr_client[0] < MAX_CLIENT);
     }
```

### 3.1.2 Client

Du coté Client il n'y a pas de changement majeur, il faut juste renseigner l'adresse ip du serveur.

## 3.2 Jeu de la vie

Nous disposons d'un nombre $N$ de clients et d'une grille de 16X16 cellules, que nous devons diviser en plusieurs blocs où chaque bloc se compose de 4 cellules.

Pour un calcul d'une génération du jeu: Le serveur doit envoyer 16 blocs au total. On considère la liste des tâche principale $LP$ qui contient le calcul des 16 blocs et selon le nombre de clients présents $nbr\_cli$, la $LP$ se divise en plusieurs sous-listes de tâches où chacune contient $nbr\_cli$ blocs à traiter. Lorsque une sous-liste de tâches est traitée, on passe à la suivante, jusqu'à épuisement des blocs de la génération courante. Une fois la génération courante totalement traitée, la suivante se met en place et ainsi de suite.

### 3.2.1 Coté Serveur

- Initialisation de la grille:

  Elle se fait par la lecture du fichier grille.init qui contient les états des cellules, le nombre de maximum des clients et le nombre de cellule par bloc. Ce fichier est généré aléatoirement par une fonction qui insère toutes les informations requises.

- Envoi des blocs:

  Les blocs sont convertis en char avec la fonction block_to_char avant l'envoi, la taille du bloc envoyé est de 4X4 cellules car on envoie le bloc et les cellules qui entourent ce bloc. Le message du bloc envoyé est de ce type: "données du bloc (16 caratères)" * "position_x du bloc" * "position_y du bloc" *.

```
    char *block_to_char(int** grille, int n, int m)
  {
    char *result=(char *)malloc(10*sizeof(char));
    char *tmp2  =(char *) malloc(sizeof(char));

    for(int i=n-1;i<n+3;i++){
      for(int j=m-1;j<m+3;j++){
sprintf(tmp2,"%d",grille[i][j]);
strcat(result,tmp2);
      }
    }

    sprintf(tmp2,"*%d*%d*",n,m);
    strcat(result,tmp2);
    free(tmp2);
    return result;
  }
```

- Réception des blocs:

  Le serveur reçoit un message sous cette forme: "données du bloc (4 caratères)" * "position_x du bloc" * "position_y du bloc" *, le message alors est convertis en bloc avec la fonction char_to_block, et est positionné dans la grille grâce à la fonction update_block.

```
   block char_to_block(char *message)
  {
    block res;
    char *buf=(char*)malloc(sizeof(char));
    int l=0;
```

```
    char *token;
    /*Contenu du block*/

    token = strtok(message, "*");

    int t= sqrt((int)strlen(token));

    int **result=(int**) malloc(t*sizeof(int*));
    for(int k=0;k<t;k++){
    result[k] =(int *) calloc (t, sizeof(int));}

    for(int i=0;i<t;i++)
    {
      for(int j=0;j<t;j++)
      {
buf[0]=token[l];
/*result[i]= message[i] -48;*/
result[i][j]=atoi(buf);
l++;
      }
     }
    res.contenu=result;
    /*pos_x*/
    token = strtok(NULL, "*");
    //res.pos_x=(int)(token[0]);
    res.pos_x=atoi(token);
    /*pos_y*/
    token = strtok(NULL, "*");
    //res.pos_y=(int)(token[0]);
    res.pos_y=atoi(token);
    return res;
  }
```

- Gestion du virus:

  Le virus apparaît toutes les 10 générations, il prend un point de la grille de manière aléatoire et marque les cellules qui sont sur sa diagonale pour que les clients tuent ces cellules.

```
   void virus(int ** grille,int p,int q)
  {/* p et q servent à initialiser srand */
    srand(q+p);
    int i,j,n,m;

    n=i=(rand())%16;
    m=j=(rand())%16;

    if (n<0)
      n=i=0;

    if(m<0)
      m=j=0;
    int quit=0;
    while(!quit){
    if ( (i >= 1) && (j >= 1 ) && (i <= 16) && ( j <= 16)){
grille[i][j]=2;
i++;
j--;
    }
```

```
    else if ( (n >= 1) && (m >= 1 ) && (n <= 16) && ( m <= 16)){
grille[n][m]=2;
m++;
n--;
    }
    else
      quit=1;
    }
  }
```

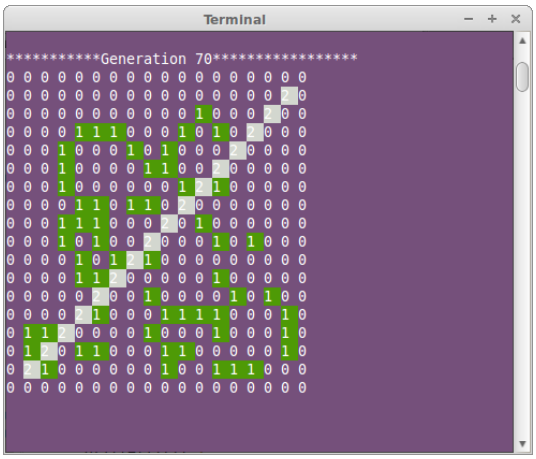



Figure 1: Apparition du virus

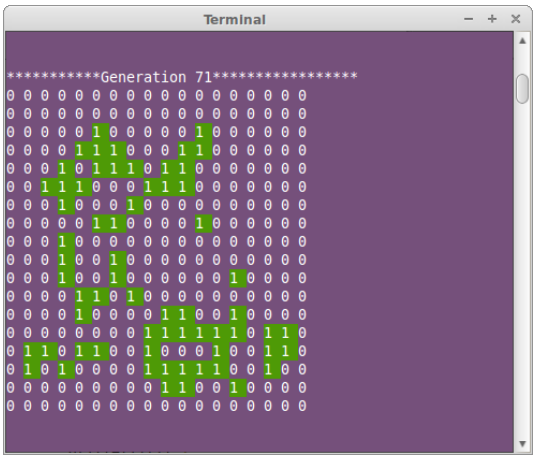



Figure 2: Génération d'aprés

- Injection de la vie:

  C'est une petite fonction qui prend plusieurs cellules de la grille aléatoirement et qui change l'état de ces cellules. Si une cellule est vivante alors elle meurt, si au contraire elle était morte elle vit. Cette fonction se déclenche quand la génération n est identique à la génération n+1 c'est à dire que soit toutes les cellules sont mortes soit qu'un ou plusieurs blocs sont stables. fonction injection de vie

```
void insert_vie(int ** grille,int p, int q)
{
  int x,y;
  for(int i=0;i<30;i++){
    srand(p+q*p);
    p++;
    x=rand()%16;
    srand(p+q*p);
    y=rand()%16;
    if(grille[x][y] != 2)
grille[x][y]=!grille[x][y];

  }
}
```

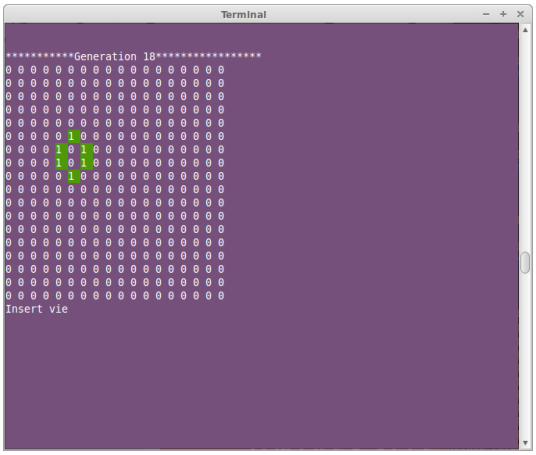



Figure 3: Etat stable

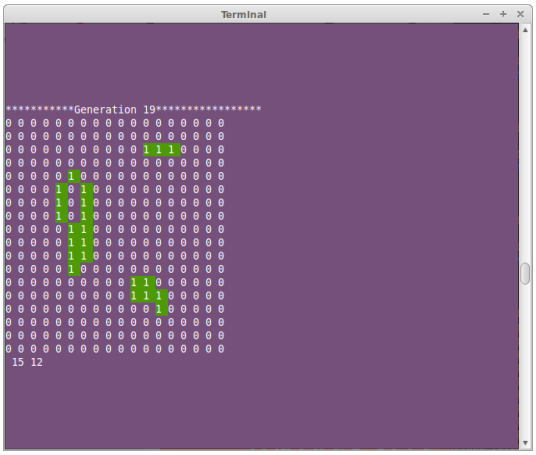
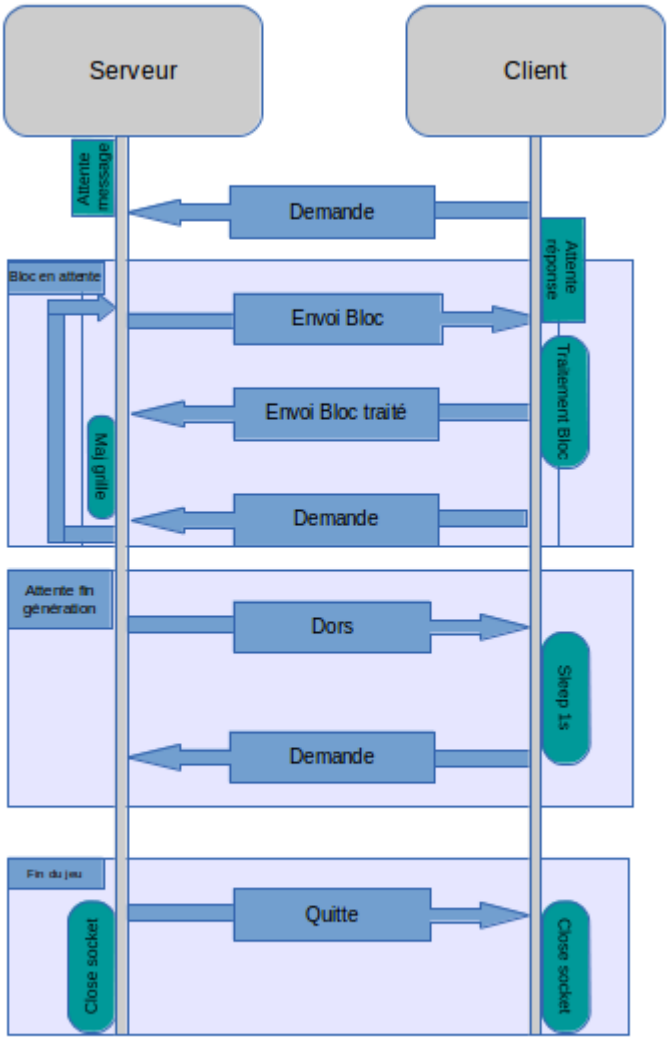



Figure 4: Insertion de vie

- Schéma de communication:
  Le schéma suivant illustre les différents types de messages entre le client et le serveur:




```
    void gestion_clients(Client *client_tab,int *nbr_client ,int **grille,int ** grille_n)
 {
char message[25];
int valide;
block tmp;
char *bloc;
int l=0;
int gen=0;
int fin = 0;
FILE * file;
file = fopen("Envoi_serveur.txt","w+");
    float temps;
    clock_t t1, t2;
    t1=0;

    while(!fin && gen < 100 ){
      fprintf(file,"\n**********Generation %d********** \n",gen);
for(int i=1; i<16;i=i+2)
{
  for(int j=1;j<16;j=j+2)
    {
```

```
      fprintf(file,"Envoi du bloc (%d,%d) de la generation %d au client %d \n",i,j,gen,l);
       /*Ecoute d'un client*/
       memset (message, 0, sizeof (message));
       //printf("**1 attente demande (%d,%d)\n",i,j);
       valide = (int) read(client_tab[l].sock,&message,sizeof(message));
       if(valide < 1)
       {
 printf("Echec réception du client %d \n",l);
 eleve_client(client_tab,nbr_client,l);
 break;
       }
       //printf("2\n");
       //strcpy(message,"demande");

       if (strcmp(message,"demande")==0)
       { /*Envoi d'un block */


memset (&message, 0, sizeof (message));
bloc = block_to_char(grille,i,j);


strcpy(message,bloc);

valide=(int) write(client_tab[l].sock,*&bloc,strlen(*&bloc));
if(valide < 1)
{
 printf("Echec Envoi du bloc (%d,%d) au client %d \n",i,j,l);
 eleve_client(client_tab,nbr_client,l);
 break;
}
/*Attente du block */

memset (&message, 0, sizeof (message));
valide = (int) read(client_tab[l].sock,&message,sizeof(message));
fprintf(file,"Réception du bloc (%d,%d) de la generation %d  du client %d\n\n",i,j,gen,l);
if(valide < 1)
{
 printf("Echec réception du bloc (%d,%d) du client %d\n",i,j,l);
 eleve_client(client_tab,nbr_client,l);
 break;
}

tmp=char_to_block(message);
update_block(tmp,grille_n);

bloc=NULL;

l=(l+1) % nbr_client[0];

       }

    }
   }
   //send_all_clients_dors(client_tab, nbr_client);
   fin = 1;
for(int i=0;i<18;i++)
  for(int j=0;j<18;j++)
```

```
    if(grille[i][j]!=grille_n[i][j]){
      fin=0;
      break;
    }
    if( gen%10 == 0)
      virus(grille_n,gen,gen*gen);
   printf("\e[1;1H\e[2J");
   printf("\n***********Generation %d**************** \n",gen);
   affiche_maj_grille( grille , grille_n);
   gen++;

if(fin){
  printf("Insert vie\n");
insert_vie(grille,gen,gen*gen);
fin=0;
}
//usleep(500000);

    }

printf("\n\n Tous les envois sont dans le fichier Envoi_serveur.txt\n\n");

fclose(file);

}
```

### 3.2.2 Coté Client

Le client suit les opérations suivantes:

1 Envoie de la demande:

Dès que le client se connecte, il envoie le message "demande" au serveur, cela signifie qu'il attend une instruction.

2 Traitement de l'instruction:

Selon l'instruction du serveur, le client analyse le message reçu :
S'il reçoit un message sous la forme "xxxxxxxxxxxxxxxx"*"x"*"x"* cela signifie que c'est un bloc, il transforme le message en un bloc, fait le traitement adéquat sur celui-ci, puis le transforme en un message sous la forme "xxxx"*"x"*"x"* qu'il envoie au serveur.
S'il reçoit le message "quitte" alors le jeu est fini, il ferme son socket et arrête de tourner.
S'il reçoit le message "dors" il dort une seconde et dès qu'il se réveille, il envoie une demande au serveur.

```
void communication(SOCKET soc)
{
  char message[20];
  int valide;
  char * rep;

  while(1){
  /* envoie demande */
    memset (&message, 0, sizeof (message));
    strcpy(message,"demande");
    valide =(int) write(soc,&message,strlen(message));

    if(valide < 1){
      printf("Echec envoi serveur !");
      exit(1);
    }

    memset (&message, 0, sizeof (message));
    printf("1 attente block\n");
    valide=(int )read (soc,&message,26);

    if(valide < 1){
    printf("Echec reception serveur !\n");
    exit(1);
    }

     if(message[0]=='1' || message[0]=='0' || message[0]=='2' || message[0]=='3')
     {
printf("Block reçu : \n");
puts(message);

block inst = char_to_block(message);
 /*TRaitement du Block*/
        inst = au_travail(inst);
rep  = block_to_char(inst);
memset (&message, 0, sizeof (message));
strcpy(message,rep);
printf("Block Traite : \n");
puts(message);
      valide= (int) write(soc,&message,12);
      if(valide < 1){
perror("Echec envoi serveur !");
sleep(1);
exit(1);
      }
     }
     if(strcmp(message,"quitte")==0)
```

```
      {
        break;
      }
      if(strcmp(message,"dors")==0)
      {
        printf("Je dors une demi seconde \n");
        usleep(500000);
      }
  }
}
```

3 Traitement du bloc:

Le bloc est reçu comme un char sous cette forme "xxxxxxxxxxxxxxxx"*"x"*"x"*.
La première partie du message qui correspond au contenu du bloc envoyé par le serveur est transformé en une matrice de 4X4 à qui on applique un traitement sur les 4 cases du milieu.
Pour chaque case, on compte le nombre de ses voisins et selon ce nombre la cellule est notée vivante ou morte.
Si la cellule contient un virus, elle meurt sans traitement préalable.
Une cellule contenant un virus ou étant morte n'est pas comptée parmi les voisins, les voisins étant forcément des cellules vivantes.

```
  block char_to_block(char *message)
  {
    block res;
    char *buf=(char*)malloc(sizeof(char));
    int l=0;
    char *token;
    /*Contenu du block*/
    token = strtok(message, "*");
    int t= sqrt((int)strlen(token));

    int **result=(int**) malloc(t*sizeof(int*));
    for(int k=0;k<t;k++){
    result[k] =(int *) calloc (t, sizeof(int));}

    for(int i=0;i<t;i++)
    {
      for(int j=0;j<t;j++)
      {
buf[0]=token[l];
result[i][j]=atoi(buf);
l++;
      }
     }
    res.contenu=result;
    /*pos_x*/
    token = strtok(NULL, "*");
    //res.pos_x=(int) (token[0]);
    res.pos_x=atoi (token);
    /*pos_y*/
    token = strtok(NULL, "*");
    //res.pos_y=(int)(token[0]);
    res.pos_y=atoi(token);
    return res;
  }
```

```
  char *block_to_char(block input)
  {
    char *result= (char *)malloc(11*sizeof(char));
    char *tmp2  =(char *)malloc(sizeof(char));

    for(int i=0;i<2;i++){
      for(int j=0;j<2;j++){
sprintf(tmp2,"%d",input.contenu[i][j]);
strcat(result,tmp2);
      }
    }
    sprintf(tmp2,"*%d*%d*",input.pos_x,input.pos_y);
    strcat(result,tmp2);
    free(tmp2);
    return result;
  }

  int count_voisin(int ** block,int i, int j)
  {
    int voisin=0;
    for(int x=-1;x<2;x++){
for(int y=-1;y<2;y++){
  if( block[i+x][j+y]==1 )
    voisin++;          //block[i+x][j+y] ;
}
    }
    if(block[i][j])
      voisin--;
    return voisin;
  }


block au_travail(block toto)
{
  int voisin;
  //int **block = toto.contenu;
  int **blocks= (int **) malloc(4*sizeof(int));

  for(int k=0;k<4;k++){
    blocks[k] = (int *)calloc( 4, sizeof(int));}


  for(int i=1;i<3;i++){
    for(int j=1;j<3;j++){

      if(toto.contenu[i][j]!=2){
voisin=count_voisin(toto.contenu,i,j);
if(voisin == 3 && toto.contenu[i][j] == 0)
{
  blocks[i-1][j-1]=1;
}
else if((voisin==3 || voisin == 2)&& toto.contenu[i][j] == 1)
{
  blocks[i-1][j-1]=1;
}
else
  blocks[i-1][j-1]=0;
      }
```

```
      else
      {
blocks[i-1][j-1]=0;
      }

    }
  }

  toto.contenu=blocks;
  return toto;
}
```

# 4 Résultats

## 4.1 Objectifs atteints

Nous avons testé le programme en faisant varier le nombre de clients afin de calculer le temps d'exécution de chaques paramètres. Nous avons remarqué qu'il y a eu une légère baisse du temps d'éxécution quand le nombre de clients augmente mais elle n'est pas très significative.
Le graphe suivant illustre les résultats obtenus:


## 4.2 Utilisation

Pour pouvoir lancer le programme, il faut les compiler avec le simple makefile fournit.
Le serveur se lance sans arguments, pour les clients il suffit de lancer le petit script "lanceur.sh" fournit dans le dossier, en ajoutant les droits nécessaires ou bien lancer chaque client dans un terminal en mettant l'adresse ip du serveur (./client X.X.X.X).
Evolution du jeu

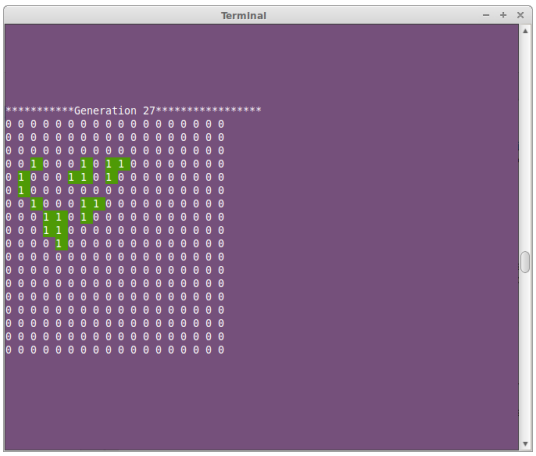



Figure 5: Génération n

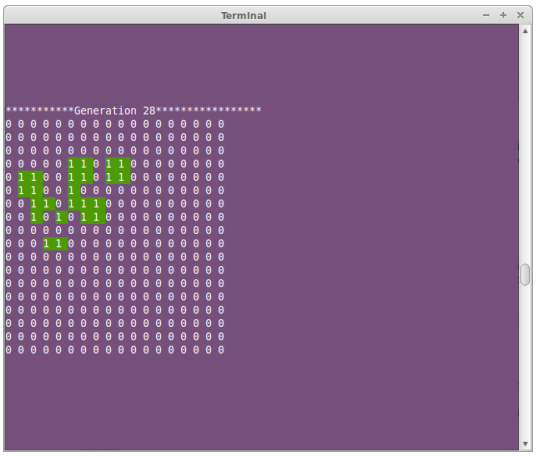



Figure 6: Génération n+1

### 4.3 Objectifs non atteints

Bien que notre programmme fonctionne correctement, nous n'avons pas pu faire en sorte qu'un autre client arrive pendant que traitement des générations est en court. Chez nous, le programme attend les clients qu'il lui faut pour commencer le travail. Par ce fait, la gestion des clients est non optimale.
L'affichage n'est pas percutant car les opérations en cours ne sont pas illustrées de façon textuelle, ce qui provoque un manque de visibilité des tâches distribuées.

## 5 Conclusion

Le projet nous a permis de comprendre les différentes architectures distribuées et de rencontrer les difficultés liées à la gestion de ces structures.
Bien que nous ne maîtrisons pas parfaitement cette architecture, les bases nous semblent acquises. Nous aurions aimé créer un affichage plus intuitif et réussir à améliorer les temps d'exécution.