Parallélisation d'un algorithme de traitement d'image avec une librairie de *threads* POSIX

Fabrice Bellet

Labo TIMC/IMAG - Institut Albert Bonniot
Domaine de la Merci
38706 La Tronche Cedex - FRANCE
Tel : (33) 04 76 54 94 10
Fax : (33) 04 76 549 549
Email : Fabrice.Bellet@imag.fr
WWW : http ://www-timc.imag.fr/sic

Résumé

Ce rapport de recherche décrit mes travaux effectués au laboratoire TIMC-IMAG pendant l'année universitaire 1998-1999. Il est constitué de deux parties. La première est une présentation simple des threads POSIX, de leurs fondements par rapport aux processus lourds Unix classiques, de leur cadre d'utilisation. Cette présentation décrira également succintement quelques notions liées à la synchronisation et au partage de ressources sur des architectures parallèles. La deuxième partie présente l'exemple de l'implémentation d'un algorithme parallèle utilisant les threads POSIX. Cet exemple est issu du domaine du traitement d'image, il est simple et suffisamment configurable pour permettre de faire des mesures de performances dans différentes conditions d'exécution. Nous mettrons enfin en évidence la nécessité de gérer avec beaucoup de précision les points de synchronisation dans un algorithme parallèle. Ce rapport permettra enfin la comparaison entre différentes architectures parallèles sous différents systèmes d'exploitation.

Remerciement

Je remercie les membres de l'équipe Apache pour le matériel qu'il m'ont permis d'utiliser, et particulièrement Alexandre Carissimi pour son aide précieuse dans l'implémentation de mes algorithmes. Je remercie également Bernard Cassagne pour la documentation qu'il m'a fournie et Catherine Cassagne pour m'avoir permis d'utiliser le serveur quadri-processeur de l'ENSIMAG pour mes mesures. Je remercie enfin Catherine Garbay dans l'équipe de laquelle j'ai effectué ces travaux.

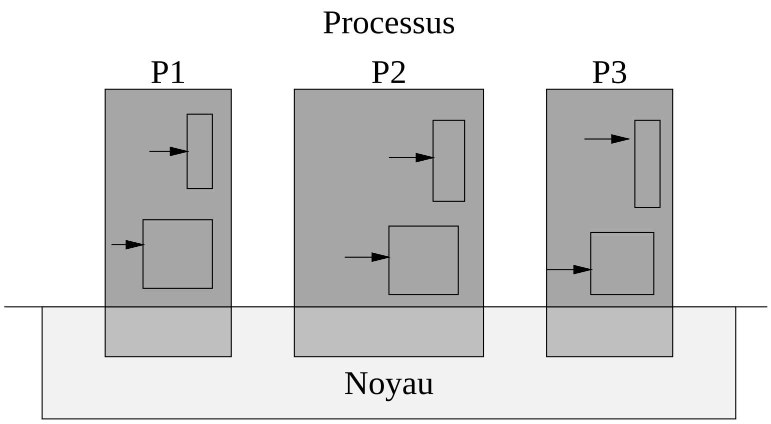

FIG. 1 – Chaque processus dispose de sa propre zone mémoire où est stocké le code du programme exécuté, la pile d'exécution, les variables globales. Chaque processus peut utiliser une quantité arbitraire de mémoire pour contenir ces informations.

1 Introduction

2 La librairie de threads POSIX

Cette partie est inpirée de deux ouvrages de référence [1] et en particulier [2] dans sa seconde édition qui apporte une vue très pédagogique du rôle des threads par rapports aux processus Unix. Ce deuxième ouvrage ayant été écrit par des développeurs de Sun, il apporte en plus un éclairage particulier sur l'implémentation très spécifique des threads POSIX choisie dans le système d'exploitation Solaris.

2.1 La notion de processus UNIX

Un processus au sens défini par UNIX est une instance d'exécution d'un programme. Il contient toutes les informations nécessaires à ce programme pour s'exécuter : un espace mémoire contenant le code du programme et la mémoire nécessaire à ses données, les valeurs des registres, ainsi que diverses informations d'ordre *administratives* qui ne nous interessent pas dans le cadre de cet article : uid, gid, priorité, temps d'exécution écoulé, masque aux signaux, ..., voir la figure 1.

Un processus UNIX dispose donc d'un ensemble de pages disjointes, allouées dynamiquement par l'appel système `brk()` dans la mémoire réelle, figure 2. Une table de correspondance permet de conserver le lien entre l'adressage virtuel effectué par le processus (le processus *voit* sa zone mémoire commencer à l'adresse zéro) et l'adressage de la mémoire réelle, figure 4.

La création de nouveaux processus se fait par l'appel système `fork()`, qui réalise une opération de clonage, par duplication de la structure de données (incluant les registres) et de l'espace mémoire du processus appelant. Les caractéristiques de fork sont donc :

- Deux processus se retrouvent à exécuter le code qui suit l'appel système à fork() : le cloneur et le cloné. Seule la valeur retournée par l'appel système permet de différencier le processus père du processus fils. Cela se justifie par le fait que les registres, et en particulier le PC [1] sont dupliqués, de même que la pile user, et la page mémoire contenant le code du programme.
- Le processus fils dispose d'une copie des variables du processus père, et il peut donc les faire évoluer indépendamment du processus père.
- Les descripteurs de fichiers sont partagés, car il s'agit de structures gérées par le noyau, que chaque processus n'accède que par un pointeur.
- La communication entre des processus est possible :
  - par des IPC (segments de mémoire partagée, figure 3, pipes, sémaphores).
  - par des signaux (asynchrone).
  - par des sockets (ie des fichiers en local).
- L'appel système `fork()` est coûteux pour plusieurs raisons :

[1] Program Counter, compteur ordinal

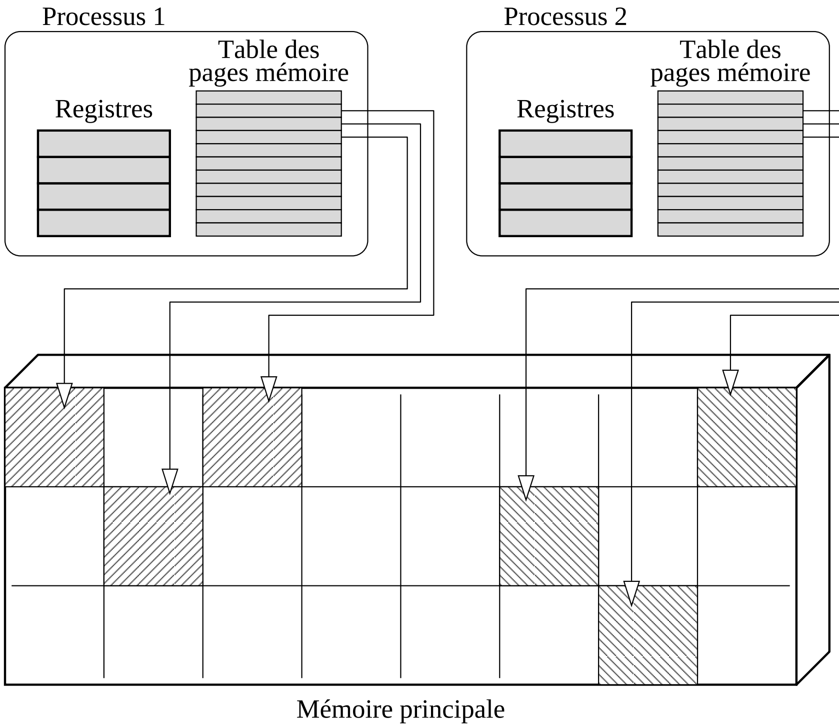

Fig. 2 – Chaque processus dispose d'une table de pages mémoire distincte définissant un ensemble de pages en mémoire principale. Ces pages sont distinctes.

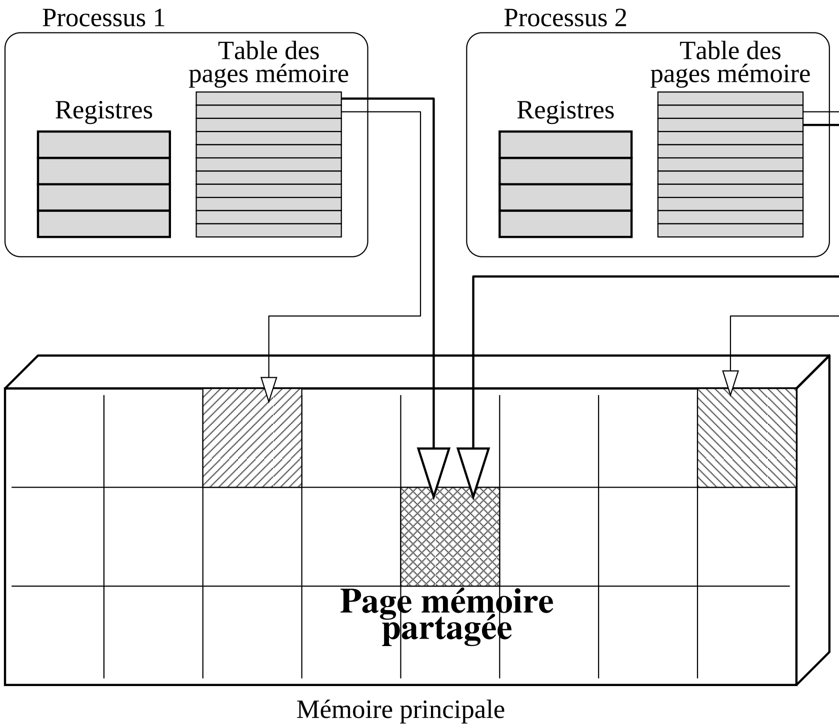

Fig. 3 – Les IPC (Inter Process Communication) permettent de définir et d'utiliser des segments de mémoire partagés entre différents processus. Dans ce cas, une entrée de la table des pages de deux processus pointe vers la même page physique en mémoire principale.

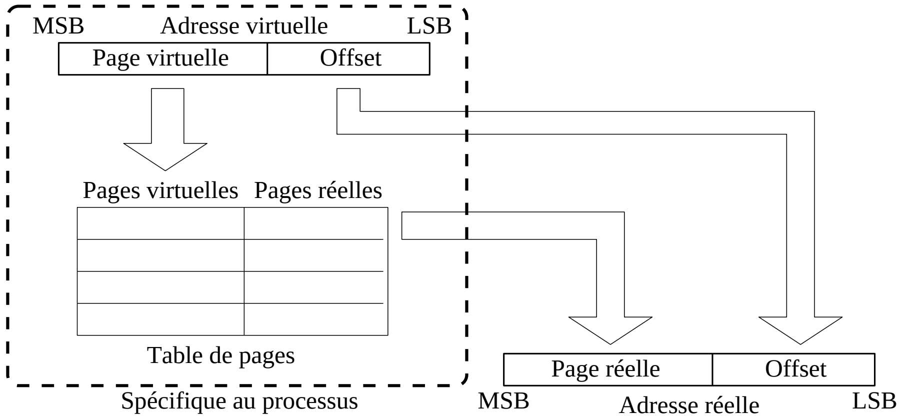

FIG. 4 – Réalisation de l'adressage de la mémoire virtuelle. Une adresse virtuelle est composée d'un numéro de page virtuelle, et d'un offset dans cette page virtuelle. Une table établit la correspondance entre le numéro de page virtuelle et de page réelle. La taille des pages est déterminée par le nombre de bits utilisés pour l'offset. Chaque processus dispose de sa propre table de pages. La conversion d'adresse est une opération cablée dans le processeur.

- C'est un appel système, il implique donc un trap, un changement de contexte, et un passage en mode noyau.
- Il duplique l'espace mémoire du processus appelant [2], demande l'allocation de nouvelles pages en mémoire réelle, qui ne sont pas forcément disponibles, et peuvent donc provoquer des défauts de pages, des accès à la mémoire secondaire pour y déposer les pages mémoires les moins récemment utilisées, etc.
- Des processus peuvent s'exécuter concurremment sur des architectures SMP [3]

Le concept de thread permet de conserver l'avantage de l'exécution concurrente en environnement SMP, tout en gagnant en légèreté d'implémentation.

2.2 La notion de thread

L'idée de base est de réaliser un fork pour lequel on ne duplique pas la mémoire. Deux types d'implémentation sont disponibles : les threads noyau ou les threads utilisateur.

2.2.1 Les threads noyau

Il s'agit d'un fork où la mémoire est commune. Le séquencement de ces threads est réalisé par le noyau, au même titre que les processus dits *lourds*.

2.2.2 Les threads utilisateur

Il s'agit généralement d'une librairie utilisateur qui gère un ensemble de piles et de registres comme cela est représenté sur la figure 5 et 6. L'état d'exécution d'un processus est défini à tout instant par :
- la valeur de ses variables globales (segment de mémoire Data).
- sa pile d'exécution (contient les pointeurs de retour des procédures appelantes, les paramètres et les variables locales).
- la valeur de ses registres (variables de travail, pointeur ordinal, registre code condition, ...)

Il suffit donc de changer de pile et de registres pour changer le contexte d'exécution du processus, ce qui peut tout à fait être effectué sans aucun appel système au noyau en utilisant un timer qui déclenche ce changement

[2]Cette affirmation n'est pas toujours vraie, et doit être pondérée. Les systèmes séparent souvent l'allocation de pages de leur "mapping" en mémoire virtuelle. Le mapping intervient après l'allocation lorsque le processus tente d'accéder effectivement à la zone mémoire allouée. C'est à ce moment qu'une page de mémoire réelle doit être "mappée" dans la mémoire virtuelle du processus, et peut éventuellement provoquer un défaut de page.

[3]Symetric Multi-Processing : une architecture multi-processeurs, où les processeurs n'ont pas de rôle dédié. Par exemple, les signaux d'interruption IRQ ne sont pas redirigés sur le même processeur. Une architecture SMP implémente une mémoire partagée par plusieurs processeurs. Chaque processeur dispose de ses propres registres, unités de calculs, et caches mémoire. Les accès aux bus (mémoire, périphériques externes) sont concurrents.

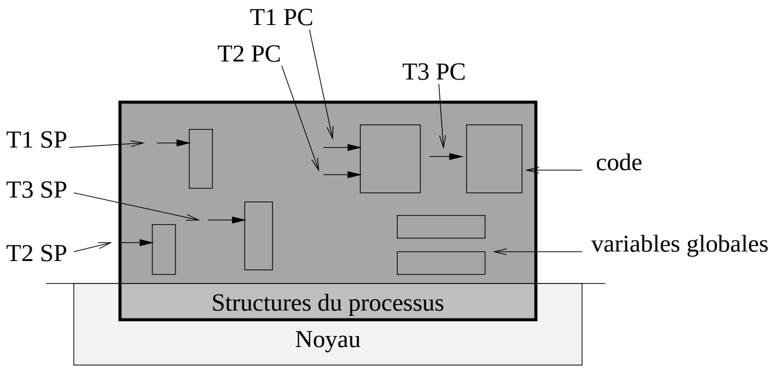

Fig. 5 – La relation entre un processus et un thread. Chaque processus abrite plusieurs threads. Ici trois threads (T1,T2,T3) disposent chacun de leur pile d'exécution (SP), de leur compteur ordinal (PC) et executent une partie du code d'un unique processus. Les deux threads T1 et T2 peuvent même par exemple exécuter le code d'une même procédure.

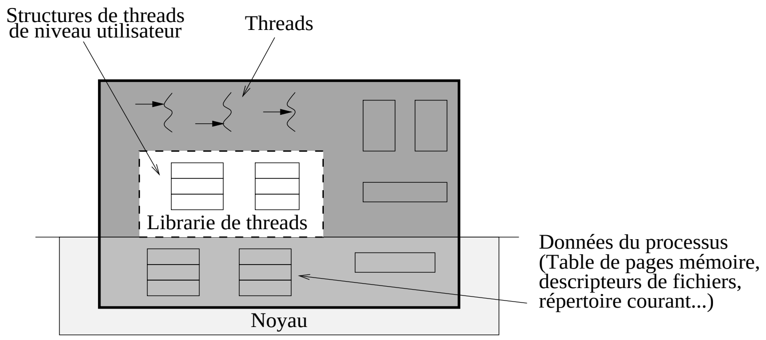

Fig. 6 – Les structures spécifiques au processus et aux threads ne sont pas au même niveau. Les structures du processus, descripteurs de fichiers, masque pour les signaux, *current working directory* sont partagées par tous les threads.

à intervalles réguliers, ou bien lorsque la librairie de threads reprend la main par l'invocation d'une de ses procédures.

La différence majeure avec les threads noyau est que le séquencement est géré par la librairie, sans passage en mode noyau, donc plus rapidement. Le changement de contexte consiste pour la librairie à recopier les registres sauvegardés dans les registres courants du processus [4]. Le séquencement d'un thread à l'autre peut donc se faire :

- à intervalles réguliers sur expiration d'un timer.
- lorsque la librairie de threads reprend la main par l'invocation d'une de ses procédures (point de synchronisation par exemple).

Plusieurs threads sont encapsulés dans un seul processus UNIX, le séquenceur du noyau n'a donc pas connaissance de l'existence de ces threads, sa granularité s'arrête au niveau du processus. Il n'y a donc pas de parallélisme possible sur une architecture SMP avec les processus légers.

Le nombre de threads n'est plus limité par le nombre de processus maximum du système, mais uniquement par la taille mémoire disponible pour stocker des piles de threads. On peut donc manipuler un grand nombre de threads sans écrouler les performances du système.

2.3 Un standard

Une API [5], les threads POSIX, rend l'utilisation des threads standardisée, quel que soit le mode d'implémentation retenu. Cette API définit un certain nombre de types, dont voici quelques exemples :

- `pthread_t` définit une structure de thread ainsi que ses attributs.
- `pthread_mutex_t` permet de manipuler des sémaphores d'exclusion mutuelle.
- `pthread_cond_t` permet de manipuler des variables de condition, soit une variable et son sémaphore d'exclusion mutuelle qui lui est associé.

Des procédures permettent de manipuler ces types, parmi lesquelles :

- `pthread_create()` permet de créer un nouveau thread.
- `pthread_exit()` termine un thread.
- `pthread_join()` permet à un thread d'en attendre un autre.
- `pthread_mutex_lock()` permet d'acquérir un sémaphore d'exclusion mutuelle.
- `pthread_mutex_unlock()` permet de libérer un sémaphore d'exclusion mutuelle.
- `pthread_cond_wait()` permet de se mettre en attente sur une variable de condition.
- `pthread_cond_signal()` permet de signaler le changement d'une variable de condition et de réveiller les processus en attente.

L'utilisateur ne connait pas directement les choix d'implémentation retenus, et ne dispose pas de moyens permettant d'avoir accès à cette implémentation. L'idée étant que les procédures fournies par l'API POSIX suffisent pour résoudre l'essentiel des problèmes de synchronisation, et que toute solution s'appuyant sur des caractéristiques dépendantes de l'implémentation est *forcément* une mauvaise solution.

Sous Linux, nous disposons entre autres de threads noyau [6]. Un appel système `clone()` spécifique a été écrit, qui se comporte comme fork, à la différence que l'on peut spécifier quels sont les composantes du processus père que l'on souhaite partager (voir `man 2 clone`). Ainsi, `fork()` est équivalent à `clone(0, SIGCLD|COPYVM)`. Une librairie de threads noyau a été écrite par Xavier Leroy (`http://pauillac.inria.fr/~xleroy/linuxthreads/`) s'appuyant sur cet appel système.

La solution choisie par Solaris est un concept hybride, utilisant la notion de LWP [7], figure 7. Un LWP est une structure qui se situe à l'interface entre l'utilisateur et le noyau. Le noyau gère un ensemble de threads de niveau noyau. Chaque LWP est géré par un thread noyau. D'autres threads noyau peuvent être utilisés pour d'autres tâches spécifiques au noyau (swap). Il s'agit de ce qui se passe du coté du sytème d'exploitation.

Du coté de l'utilisateur, il crée des threads utilisateurs dans ses processus, mais la subtilité réside dans le fait que la création d'un thread utilisateur ne génère pas *forcément* la création d'un LWP noyau associé. Un thread peut être séquencé par un LWP existant.

- Un processus peut se voir affecté à plusieurs LWP.
- Un LWP peut séquencer plusieurs threads utilisateurs.
- Un LWP est associé à un thread noyau.
- Les threads noyau sont ordonnancés sur les différents processeurs disponibles.

[4] c'est-à-dire dans les registres du processus au moment où le processus execute le code de changement de contexte.
[5] Application Programming Interface : une interface pour la programmation d'applications
[6] Mais pas uniquement : des librairies implémentant des processus légers sont également disponibles.
[7] Light Weighted Process

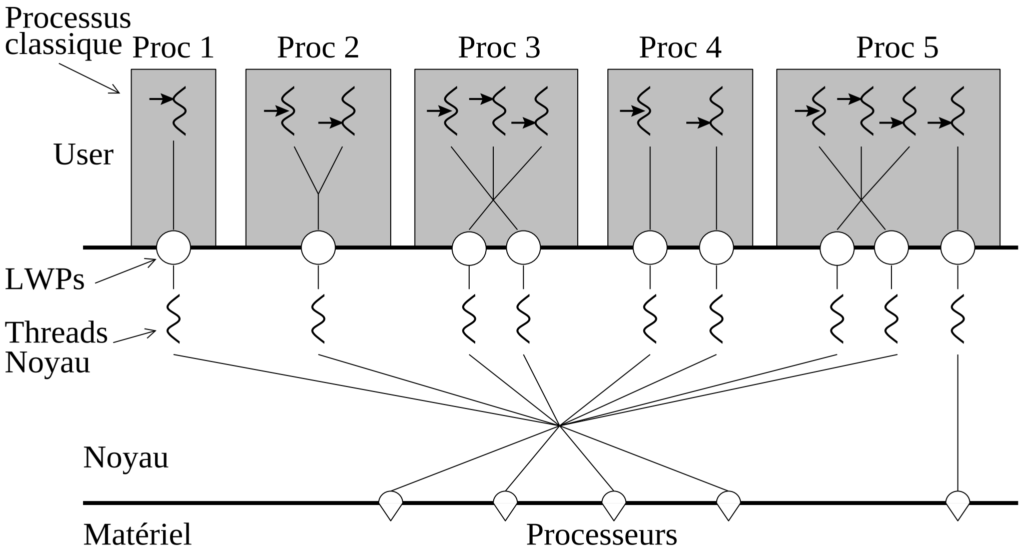

FIG. 7 – L'architecture multi-threadée de Solaris

– Un LWP, et donc un thread noyau peut restreindre son exécution à un processeur donné. [8]

2.4 Intérêt des threads

Les threads présentent des qualités qui justifient leur utilisation par rapport aux processus lourds :

– La création dynamique de threads est beaucoup plus légère que celle des processus lourds, car elle ne nécessite pas de dupliquer l'espace mémoire.
– La communication entre des threads se fait facilement par la mémoire commune du processus qui les a créés, et est plus rapide que par les moyens de communication des processus lourds.
– La réactivité par rapport aux requêtes d'un utilisateur est plus grande. Par exemple, une interface graphique ne sera pas bloquée pendant un long calcul, si interface et calcul fonctionnent dans des threads distincts si il y a préemption. Un autre exemple est d'anticiper les traitements demandés par l'utilisateur dans un thread, afin qu'ils soient disponibles lorsque l'utilisateur en fait effectivement la demande : préchargement d'images dans un diaporama. Voici un exemple de code inspiré de [1].

```
int main (int argc, char *argv[])
{
   pthread_t pic_thread;
   int fileno;
   pic_buf_t *buf;

   for (fileno=1;fileno<argc;fileno++) {
      if (fileno==1)
         /*
          * Charge la premiere image sans utiliser de thread
          */
         buf=get_next_picture((void *)argv[fileno]);
      else
         /*
          * Attend que le thread de chargement se termine
          */
         pthread_join(pic_thread,(void **)&buf);
      if (fileno<argc-1)
         /*
          * Lance un thread pour le chargement de l'image suivante
```

[8] On a intérêt à "scheduler" un thread sur le même processeur le plus longtemps possible, car cela permet de ne pas perdre les pages mémoire du cache du processeur où il s'exécutait jusqu'au moment où il est dé-schedulé.

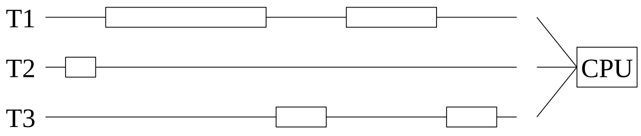

FIG. 8 – Trois threads s'exécutant concurremment sur un seul processeur.

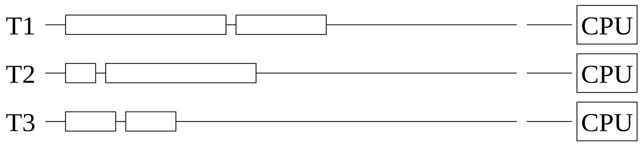

FIG. 9 – Trois threads s'exécutant en parallèle sur trois processeurs.

```
            */
           pthread_create(&pic_thread,NULL,
               get_next_picture,(void *)argv[fileno+1]);
        display_buf(buf); /* Affiche l'image courante */
        free(buf); /* libere l'image */
        if (getchar()==EOF) /* Attend une reponse de l'utilisateur */
           break;
  }
  return (0);
}

void * get_next_picture(void *arg)
{
   int fd;
   pic_buf_t *buf;

   fd=open((char *)arg,O_RDONLY);
   if (fd<0)
      return NULL;
   buf=(pic_buf_t *)malloc(...);
   while ((nbytes=read(fd,...))!=0_ {
      /* decompresse l'image dans buf */
   }
   close (fd);
}
```

- Une accélération est possible par rapport à un algorithme séquentiel, **même** sur une architecture UP [9]. Le gain pourra se faire lors des entrées sorties. En version séquentielle, le processus est bloqué sur une E/S, alors qu'en version multi-threadée, seul le thread qui demande l'entrée sortie est en attente. Les autres threads continuent à executer le code du programme.
- Une accélération est possible pour un algorithme multi-threadé entre une machine mono-processeur et une machine multi-processeur sans modification du code du programme, figure 8 et 9, sous reserve que le code de chaque thread soit *indépendant*.
- La création de plusieurs threads pour exécuter une même partie de code peut éviter des *Deadlocks* lors de mécanismes de type Client/Serveur, voir figure 10.

2.5 Contraintes des threads

La principale contrainte liée à l'utilisation des threads est la gestion des accès concurrents à la mémoire qui est laisée à la charge de l'utilisateur. Il doit donc s'assurer que les portions de code de son programme

[9]Nous notons UP pour Uni-Processor et MP pour Multi-Processor

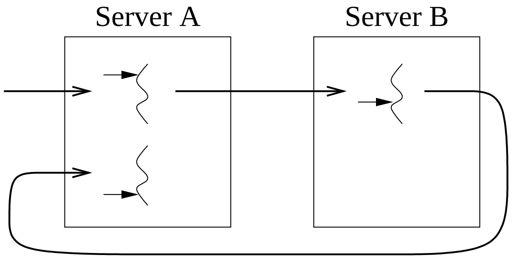

FIG. 10 – Un serveur de requêtes multi-threadé evite des *deadlocks*.

qui manipulent des variables globales en lecture comme en écriture s'exécutent en *exclusion mutuelle*. Un seul thread à la fois peut exécuter une portion de code protégée par une exclusion mutuelle.

Il faut protéger systématiquement l'accès à toute variable *globale*. A l'opposé, les variables *locales* aux procédures sont stockées dans la pile d'exécution du thread [10]. Elles sont locales à un thread et n'ont pas besoin d'être protégées.

On s'efforcera d'écrire du code *réentrant*, ce qui signifie qu'une même procédure pourra être appelée concurremment par plusieurs threads. Un thread pourra donc *réentrer* dans le corps d'une procédure alors qu'un autre thread n'en est pas encore sorti. Le programmeur avisé doit donc s'assurer que son style de programmation ne sera pas mis en défaut par cette caractéristique d'exécution. Il devra en particulier soigner les traitements qu'il effectuera sur les variables globales. Les appels systèmes n'échappent pas à cette règle. La solution de facilité consiste dans un premier temps à placer un vérou à l'entrée de tout appel système, assurant qu'un seul thread exécute du code noyau à un instant. Tout le code du noyau est donc en exclusion mutuelle, même les portions de code largement indépendantes entre elles. Cette solution est donc radicale mais inefficace. L'autre approche consiste à placer des vérous autour de portions de code noyau indépendantes entre elles : systèmes de fichiers, mémoire virtuelle, drivers de matériel, etc. On autorise ainsi plusieurs threads simultanément pour peu qu'ils fasse des appels systèmes indépendants [11].

2.5.1 Exemple

Nous allons montrer sur un exemple la nécessité de protéger des accès aux variables globales, même sur des architectures mono-processeur. Supposons que deux threads exécutent le code `a=a+1` sur une architecture mono-processeur. La traduction en pseudo assembleur de cette opération va produire le code suivant :

```
i1:  move MEM(a),Reg    ; Place le contenu de l'adresse de "a" dans un registre
i2:  incr Reg           ; Incremente le registre
i3:  move Reg,MEM(a)    ; Copie la valeur du registre en memoire
```

T1 exécute l'instruction i1, il charge `a=1` par exemple, il execute i2, puis le scheduler suspend son exécution pour faire exécuter T2. T2 exécute i1 puis i2. A cet instant Reg à la même valeur pour T1 et pour T2, puisque T1 a été interrompu avant qu'il ait eu le temps de remettre à jour la valeur de "a" en mémoire. L'ordre d'exécution de l'instruction i3 n'y changera plus rien : "a" n'aura été incrémenté qu'une seule fois bien que le code ait été exécuté deux fois.

On notera que :

- le problème n'apparait que si l'interruption du thread intervient entre l'instruction i1/i2 ou i2/i3. Les risques pour que ce cas se produise peuvent sembler faibles, en pratique cela arrive souvent. Il est indispensable de se protéger de ces effets, aussi marginaux puissent-ils être.
- détecter ce genre d'erreurs est particulièrement délicat car elles ne sont pas reproductibles. Il y a en effet peu de chances pour qu'un thread soit interrompu exactement au même endroit de son code lors de deux exécutions successives. On définit par *race condition* le cas où la valeur d'une variable dépend de l'ordre d'exécution des threads.

[10] Petit rappel : au moment de l'appel d'une procédure, un certain nombre d'informations sont empilées : les paramètres passés à la procédure, l'adresse de retour, et de l'espace est réservé dans la pile pour stocker les variables locales à la procédure. Cet espace est libéré lorsqu'elle se termine ce qui assure la contrainte de visibilité de ces variables locales.

[11] C'est une des modifications intervenue entre le noyau 2.0 et le noyau 2.2 de Linux, ce qui a permis d'accroître significativement les performances en fonctionnement SMP soit dit en passant.

2.5.2 Outils de synchronisation

Un certain nombre d'outils de synchronisation permettent de réaliser l'exclusion mutuelle entre plusieurs threads pour l'exécution de section de code critiques. L'outil de base est le *sémaphore*, crée par E.W. Dijkstra en 1960, professeur au département de Mathématiques à l'université de Technologie d'Eindhoven. Il s'agit d'un compteur couplé à une file d'attente de threads. Deux actions `P()` et `V()` sont possibles sur ce sémaphore [12] :

P() : décrémente le compteur, et bloque le thread dans la file d'attente si le compteur est inférieur à zéro. La fonction de la bibliothèque C correspondante s'appelle `sem_wait()`.

V() : incrémente le compteur, et réveille un thread de la file d'attente si le compteur vaut zéro. La fonction de la bibliothèque C correspondante s'appelle `sem_post()`.

Si le compteur du sémaphore est initialisé à une valeur $n >= 1$, cette structure garantit que au plus n threads pourront effectuer l'action P sur le sémaphore avant que le premier d'entre eux effectue l'action V. Ceci permet d'arbitrer les conflits d'allocation d'une ressource disponible en n exemplaires.

Le cas particulier où $n = 1$ résoud l'accès concurrent à une variable partagée comme celui décrit au précédent paragraphe. La ressource est ici la partie de code modifiant cette variable. On encapsule le code par des actions sur un sémaphore de la façon suivante :

```
sem_wait(&s);
a=a+1;
sem_post(&s);
```

Ces structures de synchronisation sont globales à plusieurs threads. Des primitives incrémentent et décrementent un compteur global. Donc, on pourrait supposer que l'on n'a fait que déplacer le problème de l'accès concurrent de la variable "a" vers la variable qui gère le compteur du sémaphore. En effet, que se passe-t-il si plusieurs threads exécutent concurremment un `sem_wait(&s)` sur le même sémaphore ? Le compteur ne risque-t-il pas d'être décrémenté une seule fois ? L'implémentation des sémaphores est rendue possible par l'utilisation d'une instruction machine élémentaire *Test and Set*, qui consiste à *tester* puis à *charger* une valeur en mémoire de manière atomique, c'est-à-dire sans interruption possible entre le *test* et le *set*.

```
label1      ldstub         adress,reg   ; Instruction "Test And Set" du Sparc
            compare        reg,0
            branch equal   label2
            sleep
            branch         label1
label2      return
```

- Si le mot valait 0, il passe à 1 et on considère que l'on a obtenu le lock.
- Si le mot valait 1, il reste à 1, et on n'a pas le lock.

Ce morceau de code peut être interrompu à n'importe quel moment, il assure le fonctionnement souhaité, car l'instruction `ldstub` est atomique. Un seul thread sortira de cette procédure en ayant acquis le lock.

L'exclusion mutuelle sur l'accès aux variables globales doit donc se faire à la fois en lecture et en écriture. Le dernier cas a été examiné dans l'exemple du paragraphe 2.5.1 et semble naturel. L'accès en lecture pose le même type de problème mais à un niveau plus proche du matériel. Supposons qu'une variable globale ne puisse pas être chargée depuis la mémoire en un seul accès mémoire. Supposons qu'un autre processeur entreprend de modifier sa valeur entre ces deux accès mémoire. La première partie de la variable correspondra à la valeur précédente, et la seconde partie à sa valeur courante. La valeur résultante sera incohérente et ne correspondra ni à la valeur précédente, ni à la valeur courante.

3 Modèles de programmation classique

La parallélisation d'un algorithme est en général délicate. On cherchera en général à transformer un algorithme séquentiel en utilisant un modèle qui rende sa parallélisation simple. Plusieurs modèles classiques sont facilement parallélisables :

- Le modèle **client/serveur** : le serveur lance un thread par requête d'un client (exemple classique d'un serveur Web). Ceci a pour effet de détacher le thread de traitement d'une requête du thread principal qui reste en attente de nouvelles connexions. Le serveur a une meilleure réactivité que dans sa version séquentielle consistant à mettre en attente les demandes de connexions. L'inconvénient est que le nombre de threads n'est pas limité *a priori*.

[12] "V" pour *verhogen* en Danois, et "P" pour *proberen*

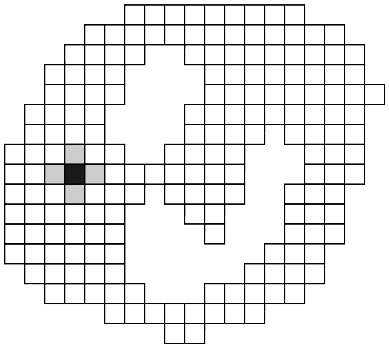

FIG. 11 – L'élément de base du remplissage est le pixel

– Le modèle **producteur/consommateur** : Un ou plusieurs threads produisent des ressources qui sont consommées par un plusieurs autres threads. Le nombre de threads maximum peut être connu.
– Le modèles des **tâches itératives** : Chaque thread exécute le même algorithme sur un jeu de données différent.

On cherchera à isoler des parties de code pouvant fonctionner avec le minimum d'interaction entre elles. Toute interaction entre deux threads nécessite que les deux threads se synchronisent. Un des deux threads devra attendre l'autre, ce qui constituera autant de temps perdu si le *scheduler* n'arrive pas à trouver un autre thread sur le processeur libéré.

En conclusion, la parallélisation d'algorithmes en utilisant une librairie de threads POSIX permet un gain en performance à la fois sur des architectures mono et multi-processeurs à mémoire partagée. Les réflexes de programmation doivent cependant s'adapter à de nouvelles contraintes. La norme POSIX implémentant les threads est assez libre et souple. Elle permet d'implémenter les outils de synchronisation classiques. En retour, le programmeur se doit d'apporter le plus grand soin à la gestion concurrente de ses variables.

4 Parallélisation d'un algorithme de remplissage

4.1 Etat de l'art

Les algorithmes de remplissage de contour fermés sont utilisés dans la plupart des logiciels de manipulation d'image et ont été abordés très tôt dans la littérature. Shani [4] et Pavlidis [3] ont proposés des algorithmes efficaces au début des années 80 au SIGGRAPH, nous allons nous en inspirer pour proposer une version plus aisément parallélisable.

Description du problème

Le remplissage d'une courbe fermée peut être vu comme l'étiquetage itératif de pixels d'une image 2D vérifiant une certaine propriété P. Le processus démarre d'un pixel initial, appelé *germe* ou *seed* dans la littérature, pour lequel la propriéte P est vérifiée. Il consiste ensuite à rajouter des pixels connexes à la zone en cours de construction, pour lesquels la propriété P est également vérifiée.

Approches utilisées

Shani a tout d'abord défini la granularité du remplissage. Il est ainsi possible de travailler au niveau :

1. du pixel, voir figure 11.

2. du segment horizontal, voir figure 12.

3. de la région régulière (*regular region* dans son article), voir figure 13.

Dans chacun de ces trois exemples, l'élément de base du remplissage apparaît en noir. Qu'il s'agisse du pixel, du segment horizontal ou bien de la région régulière, ces éléments ont la particularité de pouvoir être construit par un algorithme séquentiel. Shani définit une région régulière comme une régions qui peut être

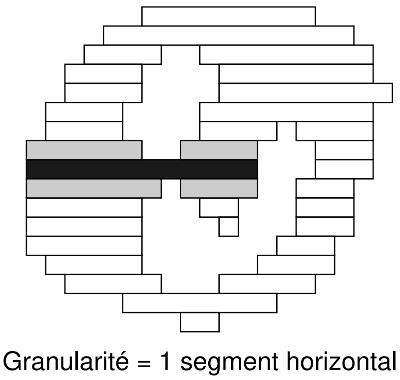

FIG. 12 – L'élément de base du remplissage est le segment horizontal

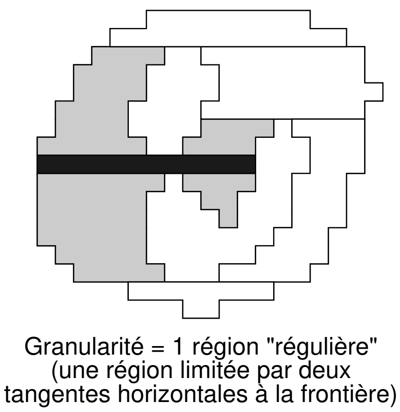

FIG. 13 – L'élément de base du remplissage est la "région régulière"

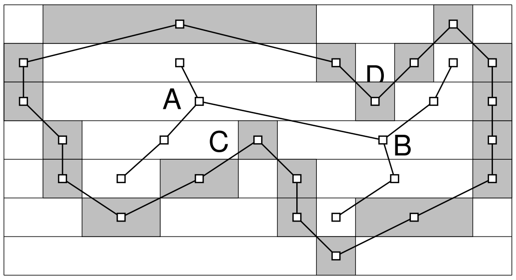

FIG. 14 – Définition des graphes d'adjacences de ligne

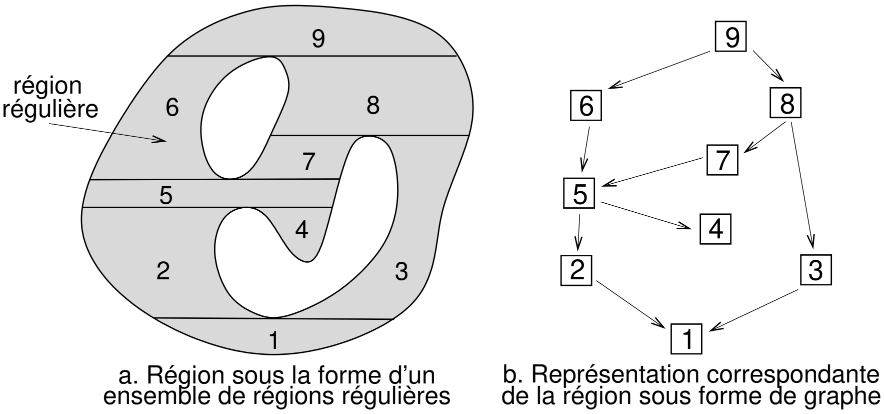

FIG. 15 – Définition du graphe d'adjacence des régions de Shani

remplie facilement par un parcours *scan* systématique de ses éléments, et qui à ce titre ne contient pas de parties concaves, ou de trous.

L'étape suivante consiste à propager la construction aux éléments qui leur sont connexes et qui apparaîssent en gris sur les figures. Un algorithme récursif vient tout de suite à l'esprit pour cette opération.

Pavlidis a formalisé le cas numéro **2.** sous la forme la forme d'un parcours de graphe [3]. Il définit la notion de graphe de lignes adjacentes (*LAG* pour *line adjacency graph*), qu'il développe aussi bien pour les pixels *intérieurs* (i-LAG) que pour les pixels définissant la frontière de l'objet à remplir (c-LAG), voir la figure 14. Chaque nœud du i-LAG représente un segment, et un arc relie deux nœuds si les segments sont adjacents [13].

Shani aboutit également à une représentation des régions à remplir sous forme de graphe orienté, acyclique, voir la figure 15. L'orientation des arcs est déduite de la position relative des deux régions associées à chaque nœud. Le problème du remplissage se transpose donc en un problème de parcours de graphe.

Pour Pavlidis, ce parcours de graphe se fait en même temps que sa construction en utilisant des primitives d'analyse très simples décrites en figure 16. Les pixels en pointillés sont ceux pour lesquels la propriété d'appartenance à la région courants n'est pas validée. Ils constituent donc ce qui formera la frontière de l'objet. On suppose qu'à cet instant de la construction, le pixel germe était situé à l'intérieur du segment délimité par les pixels $P1(x1-1,y)$ et $P2(x2+1,y)$, avec $x1 \leq x2$. La recherche des nouveaux segments (en gris clair) se fait en parcourant la bande de pixels situées en ordonnée $y-1$ (au dessus) et $y+1$ (en dessous) selon l'algorithme suivant, voir la figure 17 et 18 :

```
void segment (int x1,int x2,int y) {
   /* Trace le segment situé entre x1 et x2 */
   Draw (x1,x2,y);
   /* Parcours récursif de recherche des segments en dessous */
   x3=x1;
   {
      /* x3 est l'abscisse du 1er pixel qui vérifie la propriété
```

[13]Les deux segments sont adjacents si il existe au moins un pixel d'un segment qui est voisin d'un pixel de l'autre segment, en 4-connexité.

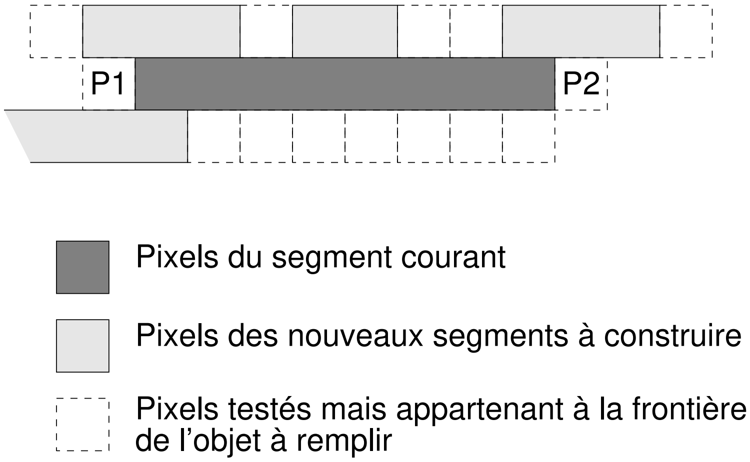

FIG. 16 – La détection des segments adjacents

```
         * dans un parcours scanline de gauche à droite de l'image en
         * partant du pixel situé sous l'extrémité gauche du segment
         * courant (fig a.)
         */
        while (!P(x3,y+1)) x3++;
        x4=x3;
        /* x4 est donc l'abscisse de l'extrémité de gauche du nouveau
         * segment. Il est nécessaire de faire un parcours vers la gauche
         * de l'image dans le cas ou P(x1,y+1)=TRUE (fig b.)
         */
        while (P(x4,y+1)) x4--;
        /* on cherche x5, l'extrémité de droite de ce même segment. Une fois
         * trouvée, on appelle récursivement la procédure sur ce nouveau
         * segment (fig c.)
         */
        x5=x3;
        while (P(x5,y+1)) x5++;
        segment (x4+1,x5-1,y+1);
        /* on continue la recherche vers la droite sur la ligne de pixels
         * de meme abscisse (figs d.,e.,f.)
         */
        x3=x5;
    } while (x3<x2);
    /* Meme principe pour les segments du dessus */
    {...}
}
```

Pavlidis utilise une pile de segments à traiter à la place d'appels récursifs. On remarque que cet algorithme implémente une parcours en profondeur d'abord du graphe i-LAG défini en figure 14. L'appel de procédure $P(x, y)$ vérifie la propriété d'appartenance du pixel (x, y) à la région que l'on souhaite remplir. Elle devra en plus vérifier que le pixel testé n'a pas été déjà étiqueté. Enfin, par souci de simplicité, les conditions aux bords de l'image ne sont pas testées.

Cet algorithme ne pose pas de difficultés, il a été implémenté avec succès dans sa version séquentielle, telle qu'elle avait été décrite à l'époque. Sa parallélisation n'avait pas été abordée. Nous proposons d'étudier cette parallélisation.

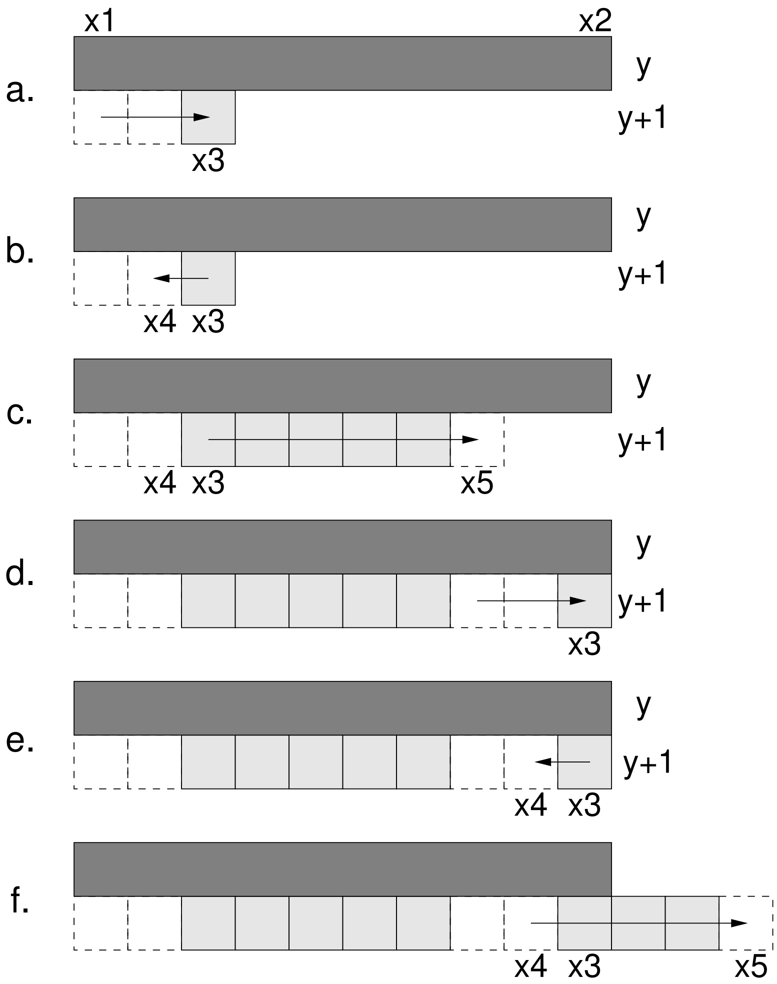

Fig. 17 – Le détail de la détection des segments adjacents (cas 1)

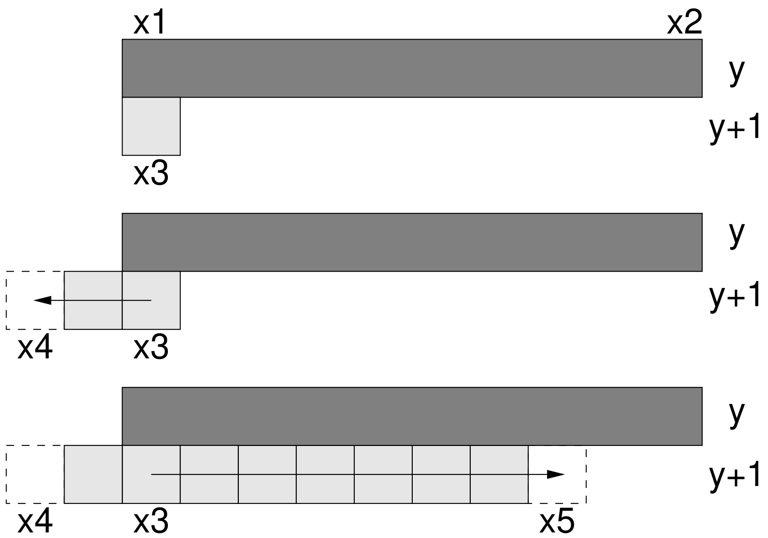

Fig. 18 – Le détail de la détection des segments adjacents (cas 2)

4.2 Parallélisation

Un algorithme sous forme récursive se prête mal à la parallélisation. Pour une procédure récursive de ce type :

```
void P() {
   some_work;
   P(); /* 1 */
   some_other_work;
   P(); /* 2 */
   some_other_work_again;
}
```

On pourrait être tenté de lancer chaque appel récursif dans un *thread* séparé et d'attendre leur terminaison par un `pthread_join()` en fin de procédure :

```
void ugly_MT_P() {
   pthread_t t1,t2;

   some_work;
   pthread_create (&t1,pthread_attr_default,(void *)&ugly_MT_P,NULL);
   some_other_work;
   pthread_create (&t1,pthread_attr_default,(void *)&ugly_MT_P,NULL);
   some_other_work_again;
   pthread_join (t1,NULL);
   pthread_join (t2,NULL);
   pthread_exit();
}
```

Le gros inconvénient d'une telle solution est que l'on ne dispose d'aucun contrôle et d'aucune borne sur le nombre de threads qui fonctionneront à un instant donné dans le système. On préférera de beaucoup se ramener à un schéma de programmation parallèle classique parmi ceux décrits au paragraphe 3, et en particulier le modèle *producteur-consommateur* peut bien s'adapter à notre problème de remplissage.

Le modèle des producteurs-consommateurs

Considérons un ensemble de threads ayant alternativement un rôle de consommateur puis de producteur (par exemple de segments). Dans l'étape de construction décrite à la figure 16, on considère que le thread courant vient de **consommer** le segment courant délimité par les pixels $P1$ et $P2$. Il effectue son étiquetage, puis il **produit** les autres segments de part et d'autre du segment courant.

Chaque thread entre donc en concurrence pour l'accès à une liste de ressources à produire et à consommer, par exemple les segments. Une ressource produite par un thread ne sera d'ailleurs pas forcément consommée par ce même thread. La seule différence par rapport au modèle classique [14] réside dans le fait que chaque thread a exactement le même rôle, alors que le modèle classique implémente un certain nombre de threads qui sont uniquement producteurs ou consommateurs. Le gros intérêt de cette modélisation sera d'avoir un nombre de threads constant et fixé dès le lancement de l'algorithme.

Chaque thread va donc implémenter un code de ce type :

```
while (1) {
   /*
    * Phase de consommation
    */
   s=GetSegment();
   DrawSegment(s);
   /*
    * Phase de production
    */
   while (condition) {
      s=FindNewSegment();

      PutSegment(s);
```

[14] et qui posera des problèmes de synchronisation par la suite.

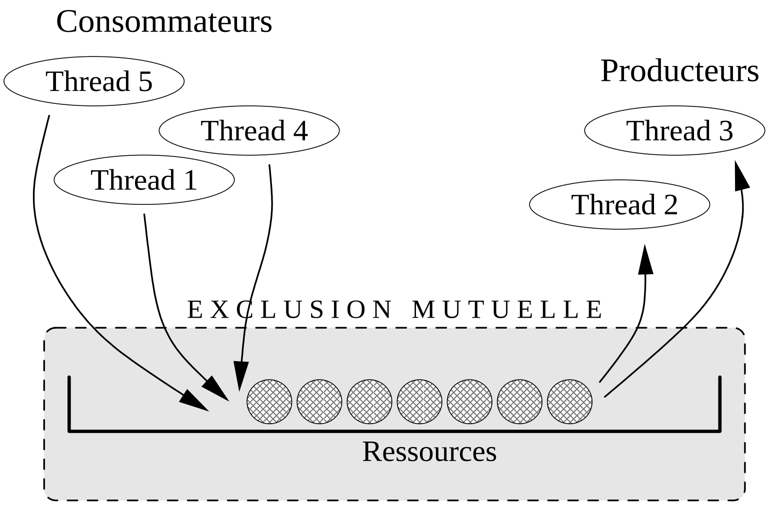

FIG. 19 – Le modèle des producteurs-consommateurs

```
    }
}
```

Plusieurs points restent à définir :

- Le choix du niveau de granularité des threads, c'est-à-dire de la sémantique associée à la ressource : pixel, segment ou région régulière. Il est évident que le choix retenu influencera la proportion du temps passé par chaque thread entre synchronisation et traitements effectués sur la ressource. Nous allons étudier le cas où la ressource est un segment horizontal.
- Le choix du mode de gestion de la liste des ressources (FIFO ou LIFO).
- La contrainte d'arrêt n'est pas triviale, et ne peut pas se limiter à tester que la liste des ressources est vide, car un thread étant alternativement producteur et consommateur, il existe une durée t pendant laquelle un thread aura pris la dernière ressource et n'en aura pas encore produit de nouvelle.
- Cet exemple introduit des contraintes de synchronisation spécifiques. Il n'est pas génant par exemple qu'un pixel soit étiqueté simultanément par plusieurs threads, car il le sera à la même valeur.

Les contraintes de synchronisation spécifiques à cet exemple

Les variables devant être protégées sont les variables globales de l'algorithme. Il s'agit des structures implémentant la liste des segments, et des structures d'images manipulées. La liste des segments ne peut être accédée en lecture ou en écriture (ajout ou suppression d'un item) que par un seul thread à la fois, figure 19.

Les structures images sont un cas un peu particulier. On distingue l'image source, sur laquelle est évaluée la propriété `P()` d'appartenance à la région. Cette image est constante et n'a donc pas besoin d'être protégée. On distingue aussi l'image des labels, dans laquelle chaque thread trace les segments à mesure qu'il les détecte (`DrawSegment(s)`). Cette écriture n'a pas besoin d'être faite en exclusion mutuelle, car un pixel ne peut passer que de la valeur 0 (fond) à la valeur 1 (région), et il n'est donc pas génant que plusieurs threads réalisent simultanément cette opération, puisque :

- deux écritures simultanées au même pixel, donc à la même adresse mémoire, ne risquent pas de corrompre la valeur écrite.
- les écritures simultanées ne se généralisent pas suffisamment pour écrouler les performances. L'idée est que des écritures simultanées introduisent un léger *overhead* qui est moins pénalisant que celui qui serait induit par l'exclusion mutuelle en écriture.

La synchronisation sur la liste des segments

La synchronisation pour l'accès à la liste des segments peut être résolue par l'utilisation de variables de conditions, figure 20. La variable de condition est un outil de synchronisation robuste permettant de rester en attente tant qu'une condition est fausse, et d'être réveillé lorsqu'elle devient vraie. Une variable de condition se voit toujours associée à un *mutex*. La variable ne peut pas être modifiée ou lue sans avoir obtenu le *mutex*. Un premier thread teste la valeur de la condition. Si celle-ci est vraie, le *mutex* est libéré et le thread continue

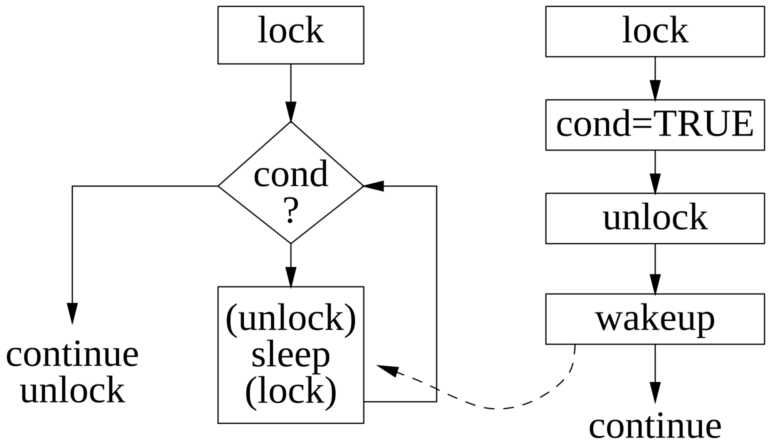

FIG. 20 – Utilisation d'une variable de condition

son exécution. Si celle-ci est fausse, le thread se met en sommeil après que le mutex ait été libéré. Un deuxième thread pourra alors acquérir le *mutex* et modifier la variable de condition. Il signale alors cette modification par un appel à `pthread_cond_signal()`, ce qui a pour effet de réveiller un des threads en attente.

Dans le cas du modèle des producteurs-consommateurs, deux variables de conditions sont nécessaires : une pour signaler que la liste des ressources est pleine et une autre pour indiquer qu'elle est vide.

```
pthread_cond_t req_producer;    /* Variables de condition */
pthread_cond_t req_consumer;
pthread_mutex_t req_lock;
short nb_atwork=0;

while (1) {
   /*
    * Consommation
    */
   pthread_mutex_lock (&req_lock);
   while (EmptyTab() && nb_atwork) {
      pthread_cond_wait (&req_consumer,&req_lock);
   }
   if (EmptyTab() && !nb_atwork) break;
   s1=GetSeg();
   nb_atwork++;
   pthread_mutex_unlock (&req_lock);
   pthread_cond_signal (&req_producer);
   DrawSeg(s1);

   ...

   /*
    * Production
    */
   pthread_mutex_lock (&req_lock);
   while (FullTab())
      pthread_cond_wait (&req_producer,&req_lock);
   PutSeg (x1,x2,y);
   pthread_mutex_unlock (&req_lock);
   pthread_cond_signal (&req_consumer);

   ...
```

```
    pthread_mutex_lock (&req_lock);
    nb_atwork--;
    pthread_mutex_unlock (&req_lock);
}
pthread_mutex_unlock (&req_lock);
pthread_cond_signal (&req_consumer);
pthread_exit(0);
```

Deux variables de conditions sont couplées.

- `req_consumer` associé à la condition `EmptyTab() && nb_at_work`
- `req_producer` associé à la condition `FullTab()`

Tout thread susceptible de modifier une de ces conditions doit donc envoyer un signal permettant de réveiller un autre thread. Un thread qui consomme un segment avec `GetSeg()` rend potentiellement la condition `FullTab()` fausse, et doit donc le signaler à d'autres threads bloqués sur `req_producer`. Inversement, un thread qui produit un segment avec `PutSeg()` change potentiellement la condition `EmptyTab()` ce qui implique de réveiller les threads bloqués sur `req_consumer`.

La variable `nb_atwork` permet de tenir le compte du nombre de threads situés dans leur exécution entre la production et la consommation. Le test de cette variable permet de sortir "proprement" de cette boucle lorsque tous les segments de la liste ont été consommés, et que plus aucun thread ne va en produire de nouveaux. Les modifications de cette variable se font sous le même *mutex* que celui associé aux deux variables de condition.

Un certain nombre de threads sont créés à l'initialisation. La liste des segments est initialement vide. Ils se bloquent donc tous sur la variable de condition `req_consumer`. Un premier segment est introduit dans la liste des ressources par le thread initial (le processus qui exécute le `main()`), qui lance enfin un `pthread_cond_signal (&req_consumer)` afin de mettre au travail un des threads fils. On notera que le nombre de threads est constant pendant toute l'exécution.

On remarquera que cet algorithme dissocie complètement la localisation des segments manipulés (ce qui correspond en fait à leur emplacement en mémoire) du thread qui les traite. Effectivement, un thread va produire un ou plusieurs segments qui seront par construction adjacents au segment qu'il viennent de consommer (donc d'étiqueter), mais le thread qui sera réveillé pour traiter ce ou ces nouveaux est quelconque.

5 Mesures expérimentales

5.1 Conditions expérimentales

Une image test 1024x1024 représentant une spirale a été construite pour tester l'algorithme, voir figure 21. Le germe du remplissage est placé au centre de l'image, dans la partie noire de l'image. La condition d'appartenance à la région que l'on souhaite remplir est simplissime : $I(x, y) = 255$, ce qui signifie que le pixel en coordonnées (x, y) est de couleur blanche. Ce test est volontairement simple pour bien montrer l'influence de la synchronisation (importante) des threads par rapport à leur temps de calcul (faible) [15].

Des mesures de temps d'exécution ont été effectuées sur différentes architectures mono- et multi-processeurs, en faisant varier pour chacune d'elles le nombre de threads lancés pour traiter cette même image test. Les différentes configurations matérielles testées sont les suivantes :

tri-alpha533 : un serveur Digital DEC avec trois processeurs Alpha à 533 MHz et 4 Mb de cache fonctionnant sous Digital Unix V4.0D.

bi-pII333 : un PC avec deux processeurs Intel PentiumII à 333 MHz tournant sous Solaris 2.5.

bi-pII412-linux20 : un PC avec deux processeurs Intel PentiumII à 412 MHz tournant sous Linux-2.0.36.

bi-pII412-linux22 : un PC avec deux processeurs Intel PentiumII à 412 MHz tournant sous Linux-2.2.5.

p133 : un PC monoprocesseur équipé d'un Pentium 133 MHz tournant sous Solaris-2.5.

pII300-linux22 : un PC monoprocesseur équipé d'un Intel PentiumII 300 MHz tournant sous Linux-2.2.5.

bi-usparc248 : un serveur Sun avec deux processeurs UltraSparcII à 248 MHz sous Solaris 2.5.

quad-pPro200 : un PC avec quatre processeurs Intel PentiumPro 200MHz sous Solaris 2.5.

Trois différentes expérimentations ont été menées en modifiant pour chacune d'elle la fonction de test d'appartenance à la région que l'on souhaite remplir, sans que cela ne modifie fondamentalement la topologie de la

[15] Cette condition peut être rendue arbitrairement complexe, ce qui permet d'augmenter (artificiellement) le temps de calcul par rapport au temps de synchronisation.

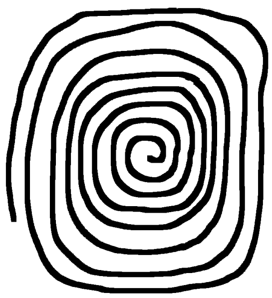

FIG. 21 – Image test 1024x1024

région résultante que l'on va remplir (en termes de nombre de segments étiquetés). Cela consiste simplement à moyenner localement la valeur du niveau de gris de l'image $I(x, y)$:

$$I(x,y) = 255 \tag{1}$$

$$\frac{1}{9}\sum_{i=x-1}^{i=x+1}\sum_{j=y-1}^{j=y+1} I(i,j) \geq 128 \tag{2}$$

$$\frac{1}{25}\sum_{i=x-2}^{i=x+2}\sum_{j=y-2}^{j=y+2} I(i,j) \geq 128 \tag{3}$$

Ces trois fonctions effectuent respectivements :
- 1 accès mémoire, une comparaison entière.
- 9 accès mémoire, 8 additions entières, 1 division en flottant, une comparaison en flottant.
- 25 accès mémoire, 24 additions entières, 1 division en flottant, une comparaison en flottant.

5.2 Analyse des résultats

Les courbes de temps d'exécution pour les trois expérimentations sont données dans les figures 22, 23 et 24.

Chaque courbe correspond à une architecture, et représente en ordonnée le temps réel [16] d'exécution en fonction du nombre de threads créés. Sur une architecture SMP, on peut s'attendre à un gain de performances lorsque l'on augmente le nombre de threads jusqu'à atteindre le nombre de processeurs, pour ensuite avoir une stabilisation (au mieux) ou bien une légère augmentation du temps d'exécution relatif à l'augmentation du nombre de synchronisations.

test1

Plusieurs choses sont remarquables dans ce graphique :
- Le temps mesuré est le *temps réel*, sur des machine multi-processus. Les mesures sont donc susceptibles d'êtres perturbées par des processus d'autres utilisateurs qui ne sont pas sous notre contrôle et par les démons système. Nous avons lancé ces expérimentations à des périodes où les machines n'avait pas d'autre processus utilisateurs en cours pour nous abstraire au maximum de ces impondérables. L'influence des démons système est marginale.
- Toutes les machines à l'exception du *tri-alpha533* ont un profil de courbe qui est conforme à ce qui est attendu. A savoir que le temps d'exécution diminue d'abord jusqu'à ce que le nombre de threads égale le nombre de processeurs, pour ensuite augmenter de façon plus ou moins importante ensuite. Le cas du *tri-alpha533* est d'ailleurs problématique, et nous y reviendrons plus tard.

[16]Le temps réel est la mesure pertinente. Les deux autres mesures de temps disponibles sont le *temps user* et le *temps système* qui donnent le temps passé par chaque thread à exécuter respectivement du code user et du code noyau. L'addition de ces temps sur le nombre total de threads ne donne pas une mesure utilisable car elle ne tient pas compte du fait que des threads ont fonctionné en parallèle.

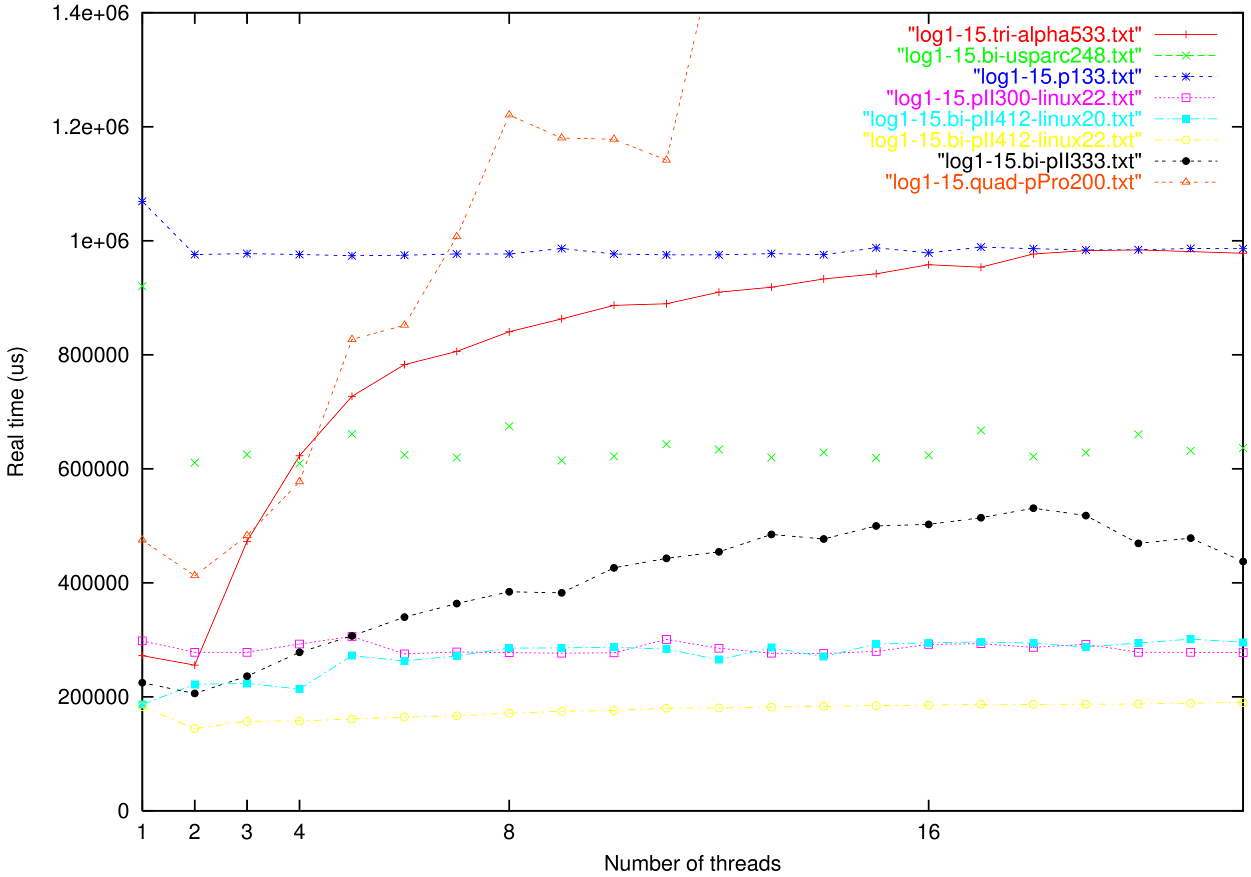

FIG. 22 – Mesure du temps d'exécution en fonction du nombre de threads (test1)

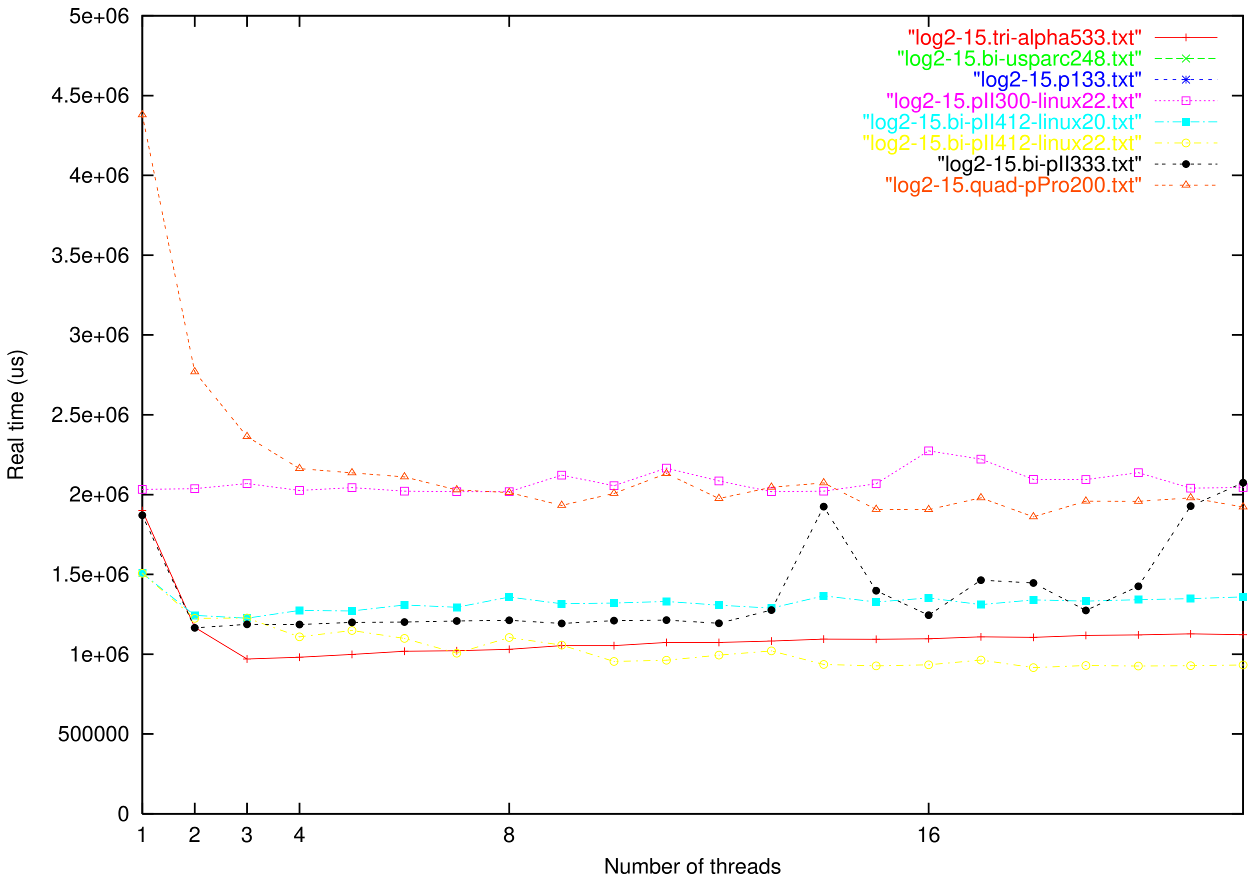

FIG. 23 – Mesure du temps d'exécution en fonction du nombre de threads (test2)

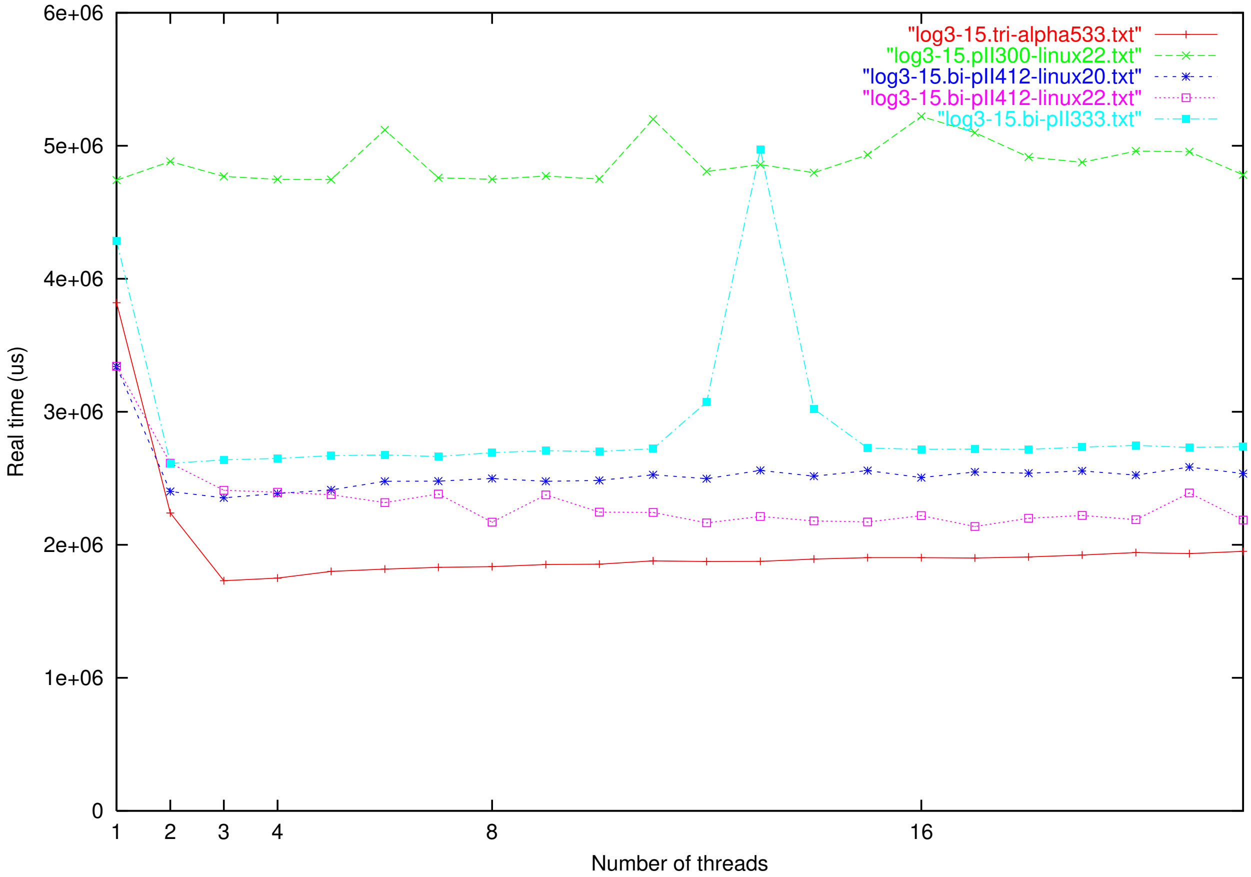

FIG. 24 – Mesure du temps d'exécution en fonction du nombre de threads (test3)

- Les machines sous Linux semblent mieux supporter un nombre de threads important que des machines équivalentes sous Solaris. Cela se traduit par des courbes qui partent pratiquement de la même valeur, mais dont les pentes sont sensiblement différentes, *bi-pII412-linux22* et *bi-pII333* sous Solaris.
- La librairie de threads POSIX n'introduit pas de gros surcoût sur les machines mono-processeurs testées, *p133* sous Solaris et *pII300-linux22*.
- La différence entre le noyau 2.0 et le noyau 2.2 de Linux est sensible entre *pII412-linux20* et *pII412-linux22*. Dans cette deuxième configuration, le gain de performance apparait entre l'exécution avec un et deux threads. Ensuite, l'augmentation du nombre de threads (et donc leurs synchronisations mutuelles) pénalise beaucoup moins le temps d'exécution avec le noyau 2.2 [17].

test2 et test3

Les échelles de temps des figures 23 et 24 ont changé par rapport à la figure 22. Et ces courbes permettent de replacer la configuration *tri-alpha533* dans une position plus en accord avec son prix. On observe une décroissance très nette jusqu'à 3 threads pour ensuite augmenter à nouveau. Les machines ayant des temps d'exécution prohibitifs pour ces deux tests, volontairement plus calculatoire, ne figurent pas sur ces courbes.

Analyse

On peut légitimement s'étonner d'une telle proximité des performances entre une configuration de type PC-Linux relativement modeste, et un serveur Alpha environ 20 fois plus onéreux, figure 24.

Il aurait été très instructif de valider cet algorithme sur deux configurations supplémentaires, qu'il ne m'a pas été possible de trouver :

- Un serveur de processeurs Alpha fonctionnant sous Linux, afin de mesurer l'influence relative de l'architecture matérielle et logicielle dans les performances mesurées précédemment.
- Une configuration de processeurs Intel gonflée (quadri-Xeon 500MHz) sous Linux pour mesurer quels gains l'architecture choisie par Intel permet d'obtenir.

Nos mesures sont évidemment incomplètes puisqu'il n'a pas été tenu compte du compilateur et des optimisations spécifiques à un processeur qu'il pouvait produire. Dans tous les cas, le compilateur utilisé est `gcc` dans une version allant de 2.6.3 à 2.8.1. Des essais ont été effectués sur Digital Unix (*tri-alpha533*) en utilisant le compilateur DEC propriétaire, mais les résultats n'étaient pas sensiblement différents. Pareillement, l'algorithme a été testé sur cette machine en utilisant l'interface propriétaire de threads fournie par Digital, CMA [18] sans que cela apporte un quelconque gain de performances. Nous n'avons pas encore trouvé d'explication satisfaisante à la contre-performance de cette machine. Il aurait été enfin utile de réaliser des mesures sur des images test plus "pathologiques" présentant davantage de discontinuités, et donc davantage de "parallélisabilité".

5.3 Optimisation

Fondements

Il est important de se pencher plus finement sur la répartition du travail entre les différents threads du système pour comprendre en quoi le *test1* constitue un cas de synchronisations extrême. En fonction de la configuration de l'image, évaluons pour un thread donné le rapport entre la quantité de ressources consommées et produites. Ceci permettra de déterminer si les threads passent du temps à attendre que la liste pleine se vide ou que la liste vide se remplissent. Dans l'exemple de la figure 25, chaque segment dispose d'exactement deux segments adjacent, à l'exception du segment situé à la jonction de la région. En supposant que l'algorthme soit initialisé par le segment situé en haut à gauche dans la branche 1, un unique thread va travailler tant que le segment de jonction ne sera pas atteint : pour un segment consommé, un seul segment sera produit. Et ceci quel que soit le nombre de threads lancés initialement. Tous les autres resteront bloqués sur la variable de condition de la liste vide.

Lorsque le segment de jonction est atteint, le thread qui le gère va produire deux nouveaux segments [19], ce qui va permettre de donner du travail à un autre thread en plus de lui-même. Ce mécanisme se généralise sur toutes les autres discontinuités de la régions. On arrive finalement à deux filières de traitement parallèle,

[17] Un gros travail a été fait par les développeurs du noyau Linux afin de favoriser le parallélisme du code du noyau. Dans les version 2.0 du noyau, un seul thread à la fois pouvait exécuter du code noyau, alors que dans les version 2.2, le vérrouillage du code est plus modulaire, et plusieurs threads peuvent exécuter concurrement du code noyau, pourvu qu'il s'agisse de parties de code indépendantes.

[18] Digital's Proprietary Interface to DECthreads. Cette interface est considérée comme obsolete dans Digital Unix V4.0D, elle continuera d'être maintenue, mais ne sera plus upgradée.

[19] Un segment déjà marqué n'est pas réintroduit !

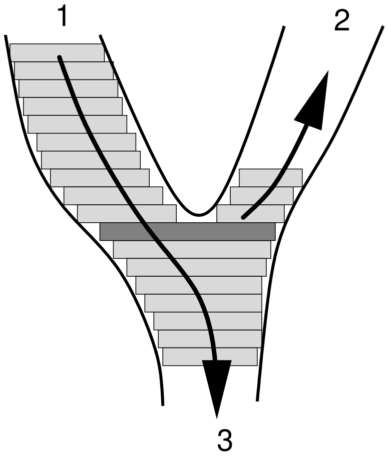

FIG. 25 – Capacité de parallélisation

représentées par les deux flèches sur la figure 25 [20]. Rien ne sert d'introduire plus de deux threads pour traiter cet exemple, quel que soit le nombre de processeurs dont on dispose. La limitation au degré de parallélisme est structurellement liée à l'image.

On arrive à la remarque que pour des régions aux formes régulières, la production de segments est dans la grande majorité des cas égale à la consommation, et supérieure seulement exceptionnellement. Augmenter le nombre de threads n'augmentera pas le degré de parallélisme, mais le diminuera au contraire, car cela ne fera qu'au contraire augmenter le nombre de threads en attente de ressources à traiter.

L'autre remarque importante consiste à évaluer les temps de calcul par rapport aux temps de synchronisation. Sur l'image traitée, le remplissage nécessite 7482 segments, pour un temps d'exécution de 200 ms environ pour les configurations rapide, et qui donne un temps de traitement de l'ordre de 25 us par segment (en configuration mono-processeur), pendant lequel le thread devra vérrouiller par deux fois un *mutex* pour accéder à la liste des segments. Le temps de calcul est donc très court entre les synchronisations [21].

Optimisation

L'objectif pour un thread qui vient de traiter un segment est de ne pas placer dans la liste des ressources le premier segment nouveau qu'il va trouver, mais seulement les segments excédentaires qu'il ne pourra pas lui même traiter au cours de l'itération suivante. Chaque thread se réserve donc le segment suivant sans avoir besoin d'entrer en concurrence pour en obtenir un nouveau à chaque itération.

Ainsi dans l'exemple de la figure 25, le même thread [22] va traiter toute la filière supportée par la flèche de gauche. Et le seul moment où un nouveau segment sera introduit dans la liste des ressources sera lorsque le segment de la jonction sera traité. Il initiera alors une nouvelle filière de traitement par un autre thread. La figure 26 montre la décomposition en régions élémentaires ainsi obtenue sur l'exemple du paragraphe 4.1. Il est à comparer à la granularité des figure 12 et 13. Ces éléments définissent les plus grandes zones pouvant être traitées de bout en bout par le même thread. Ces régions sont plus grandes que celles de la figure 13, nous les avons donc nommées “régions régulières étendues”.

Le détail du code ne présente pas d'intérêt particulier lorsque le principe a été donné. Le gain de performances apparaît sur la figure 27 où l'on note que la nouvelle implémentation est effectivement plus efficace, le gain est de 40 % pour le passage de un à deux threads, puis de seulement 8 % pour le passage de deux à trois threads. On retrouve le temps d'exécution avec un seul thread lorsque l'on en initialise 40.

[20]Ce qui n'est pas rigoureusement exact car rien ne garantit que deux segments adjacents seront traités par le même thread puisque ces segments sont réintroduit dans la liste des ressource. L'appel à *pthread-cond-signal* réveille un thread quelconque, et la norme POSIX ne nous aucun contrôle sur le choix de ce thread

[21]Si l'on considère un processeur à 300 MHz, qui exécute des instructions machine en 20 cycles en moyenne, le temps de 25 us représente approximativement 400 instructions machine, soit très peu de code.

[22]C'est cette fois-ci rigoureusement exact.

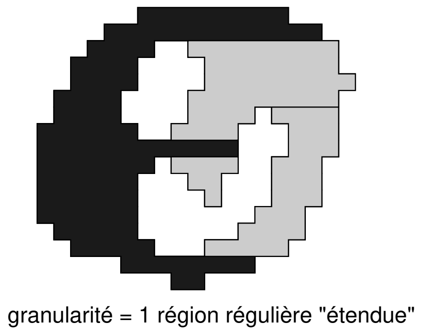

FIG. 26 – L'élément de base du remplissage est la "région régulière étendue"

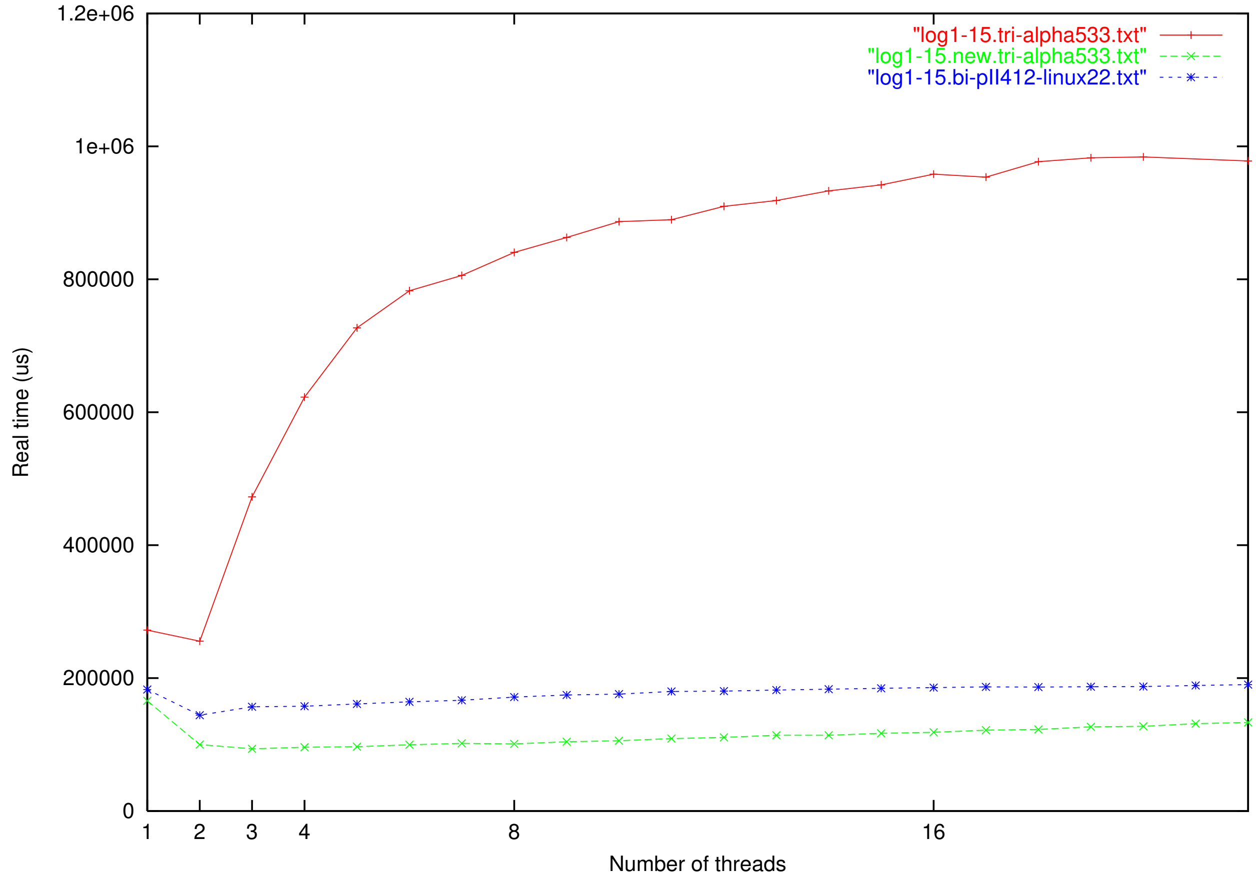

FIG. 27 – Mesure du temps d'exécution en fonction du nombre de threads (test1 optimisé)

6 Conclusion

Ce rapport de recherche décrit mes travaux effectués au laboratoire TIMC-IMAG pendant l'année universitaire 1998-1999. Son objectif était dans un premier temps de donner une présentation simple et pédagogique des threads POSIX, de leurs fondements, de leur cadre d'utilisation et de leurs limites, et dans un deuxième temps d'étudier leur fonctionnement sur un exemple réaliste mais suffisamment simple pour permettre des ajustements simples. Un exemple issu du domaine du traitement d'image a été choisi : un algorithme de remplissage de forme.

Cet algorithme a permis de tester et d'évaluer différents choix algorithmiques. Il a été testé sur une palette assez large de machines Unix, et a ainsi mis en lumière ce qui apparaît être des faiblesses sur certaines configurations.

Références

[1] S. Kleiman, D. Shah, and B. Smaalders. *Programming with threads.* SunSoft Press, 1996.

[2] B. Lewis and D.J. Berg. *PThreads Primer : a guide to multithreaded programming.* SunSoft Press, 1996.

[3] T. Pavlidis. Contour filling in raster graphics. *Computer Graphics*, 15(3) :29–36, 1981.

[4] U. Shani. Filling regions in binary raster images : a graph-theoretic approach. In *Proceedings SIGGRAPH*, pages 321–327, 1980.