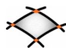
Archi'2015, Lille,
8-12 juin 2015

La parallélisation par le matériel.

Bernard Goossens, David Parello, Katarzyna Porada, Djallal Rahmoune

Université de Perpignan Via Domitia
DALI-LIRMM



Sommaire.

1. Le processeur du futur : CPU ? GPU ? autre chose ?
    - La loi de Moore.
    - Qu'a-t-on fait de nos transistors ?
    - Que va-t-on faire de nos transistors ?
2. La parallélisation.
    - La séquentialité du C.
    - La parallélisation par l'OS : exemples de *pthread* et *MPI*.
    - La parallélisation déterministe.
3. La parallélisation par le matériel.
    - Un cœur à exécution spéculative (cœur actuel).
    - Un cœur parallélisant (cœur du futur).
    - La parallélisation matérielle sur un exemple.
    - La lecture du code en parallèle.
    - Le renommage et l'exécution parallèles.
    - Le retrait parallèle.
4. Conclusion.

10 milliards de petits transistors ... et moi, et moi, et moi.

Le processeur du futur : CPU ? GPU ? autre chose ?

La loi de Moore.

La loi de Moore : loi de croissance.

- La loi de Moore est une prévision de croissance : c'est une feuille de route pour guider les industriels dans leurs choix stratégiques.
- La loi de Moore ne dit pas : la finesse de gravure sera divisée par 2 tous les 4 ans.
- La loi de Moore dit : le nombre de transistors sur un circuit intégré doublera tous les deux ans.

La loi de Moore : plusieurs sources de croissance.

- Pour maintenir une telle croissance exponentielle, la piste majeure pendant 45 ans (1971-2015) a été la réduction du pas (*pitch*) : division par 2 tous les 4 ans.
- Pour continuer, on peut encore diviser le pas par 2 (aujourd'hui 14nm, puis 7nm, puis 4nm ...).
- On peut aussi augmenter la surface du plan (mais modestement), voire exploiter une troisième dimension.

La loi de Moore : possiblement vraie encore 20 ans.

- La loi de Moore est une spéculation sur l'inventivité de l'homme. Il y a peu de risque à parier sur un maintien de la croissance mais beaucoup de risques à parier sur un maintien de la croissance exponentielle, en tous cas au delà de 20 ans.
- Toutefois, un facteur 100 sur la croissance est probable dans les 15 prochaines années et un facteur 1000 n'est pas impossible dans les 20 prochaines années.

Qu'a-t-on fait de nos transistors ?

Qu'a-t-on fait de nos transistors ? (1/5 : largeur de mot)

- 1971 : 4004, 12mm2, 2300 transistors, 15V, 1W, 4 bits, 10μm, 0.74Mhz, 16-pin.
- 1972 : 8008, 14mm2, 3500 transistors, 5V, 1W, 8 bits, 10μm, 0.5Mhz, 18-pin.
- 1974 : 8080, 20mm2, 6000 transistors, 5V, 1.5W, 16/8 bits, 6μm, 2Mhz, 40-pin.
- 1978 : 8086, 33mm2, 29000 transistors, 5V, 2.5W, 20/16 bits, 3μm, 5Mhz, 40-pin.
- 1982 : 80286, 49mm2, 134000 transistors, 5V, 3.3W, 24/16 bits, 1.5μm, 6Mhz, 68-pin.
- 1985 : 80386DX, 104mm2, 275000 transistors, 5V, 2W, 32/32 bits, 1.5μm, 12Mhz, 132-pin.

Qu'a-t-on fait de nos transistors ? (2/5 : flottant + cache)

- 1989 : 80486DX, 173mm2, 1.185M transistors, 5V, 5W, 32/32 bits + L1-8KB + FPU, 1μm, 25Mhz, 168-pin.
- Loi de Moore : 1971-1990, 20 ans, facteur 1000, 2K-2M.

Qu'a-t-on fait de nos transistors ? (3/5 : pipeline + vecteur ; course aux Mhz)

- 1993 : Pentium (P5), 294mm2, 3.1M transistors, 5V, 15W, 32/64 bits + 2*L1-8K+8K + in-order pipeline, 0.8μm, 60Mhz, 273-pin.
- 1995 : PentiumPro (P6), 307mm2, 5.5+15.5M transistors, 3.3V, 32W, 36/64 bits + 2*L1-8K+8K + L2-256K + o-o-o pipeline, 0.35+0.6μm, 180Mhz, 387-pin.
- 1997 : PentiumII (P6), 195mm2, 7.5M transistors + L2, 2.8V, 35W, 36/64 bits + 2*L1-16K+16K + L2-512K + 64bits-vectors, 0.35μm, 233Mhz, 242-pin.
- 1999 : PentiumIII (P6), 128mm2, 9.5M transistors + L2, 2V, 40W, 36/64 bits + 2*L1-16K+16K + L2-512K + 128bits-vectors, 0.25μm, 450Mhz, 242-pin.
- 2000 : PentiumIV (NetBurst), 217mm2, 42M transistors, 1.7V, 52W, L2-256K + superpipeline + tracecache, 180nm, 1300Mhz, 423-pin.
- 2003 : PentiumIV-HT (NetBurst), 145mm2, 55M transistors, 1.5V, 66W, L2-512K + hyperthreading, 130nm, 2.4Ghz, 478-pin.

Qu'a-t-on fait de nos transistors ? (4/5 : cœurs)

- 2006 : Core2-Duo (Core), 111mm2, 167M transistors, 0.8V, 65W, L2-2MB + 2 cores + 64b processor, 65nm, 2.1Ghz, 775-pin.
- 2008 : Corei7 (Nehalem), 263mm2, 731M transistors, 0.8V, 130W, L3-8MB + 4 cores, 45nm, 3.2Ghz, 1366-pin.
- Loi de Moore : 1971-2010, 40 ans, facteur 1M, 2K-2G.

Qu'a-t-on fait de nos transistors ? (5/5 : Xeon = cœurs, Corei7 = GPU)

- 2012 : Xeon-E7 (SandyBridge), 416mm2, 2.27G transistors, 0.8V, 130W, 8*64KB-L1 + 8*256KB-L2 + L3-20MB + 8 cores + 256bits-vectors, 32nm, 2.7Ghz, 2011-pin.
- 2014 : Xeon-E5v3 (Haswell), 661mm2, 5.56G transistors, 0.65V, 145W, 18*64KB-L1 + 18*256KB-L2 + L3-45MB + 18 cores, 22nm, 2.3Ghz, 2011-pin.
- 2015 : Corei7-5650U (Broadwell), 133mm2, 1.9G transistors, 0.8V, 15W, L3-4MB + 2 cores + GPU, 14nm, 2.2Ghz, 1155-pin.

Que va-t-on faire de nos transistors ?

Que va-t-on faire de nos transistors ? (1/4 : CPU 30 cœurs, GPU 5000 cœurs)

- 2016 : CPU : Cannonlake, 10nm, ? ? transistors (Moore : 16Gt).
- 2016 : GPU : Pascal, 16nm, ? ? transistors (Moore : 16Gt).
- 2019 : 7nm ? 40G transistors ?
- Loi de Moore : 1971-2030, 60 ans, facteur 1G, 2K-2T.

Que va-t-on faire de nos transistors ? (2/4 : gros cœur rapide vs petit cœur lent ?)

- 1 cœur o-o-o avec 64K-L1 + 256K-L2 + 8DP/cycle = 160M-200M transistors.
- 1 cœur in-order single-issue avec 16K-L1 = 3M transistors.
- 1MB-L3 = 50M transistors.

Que va-t-on faire de nos transistors ? (3/4 : gros caches L3 vs *multithreading*)

- GPU K110 = 7.1Gt, 15 SMX + 1.5MB-L2 => 1 SMX = 7000/15 = 470Mt.
- 1 SMX = 110KB-L1 (5Mt) + 64K-32b-RF (12Mt) + 192 SP (ou 64 DP) (2.5Mt/cœur-SP).
- Xeon-E5v3 = 5.6Gt, 18 cores (3.4Gt, 190Mt/cœur) + 45MB-L3 (2.2Gt).

Que va-t-on faire de nos transistors ? (3/4 : gros caches L3 vs *multithreading*)

- GPU K110 = 7.1Gt, 15 SMX + 1.5MB-L2 => 1 SMX = 7000/15 = 470Mt.
- 1 SMX = 110KB-L1 (5Mt) + 64K-32b-RF (12Mt) + 192 SP (ou 64 DP) (2.5Mt/cœur-SP).
- Xeon-E5v3 = 5.6Gt, 18 cores (3.4Gt, 190Mt/cœur) + 45MB-L3 (2.2Gt).

- Pourquoi un si grand L3 dans un CPU ? Parce qu'un cœur rapide lit beaucoup de code et de données et que l'écart de vitesse avec la mémoire externe est très grand : on réduit la latence par les caches.
- Pourquoi pas de L3 dans un GPU ? Parce qu'on lit peu de code (instructions SIMD) et des données à forte localité et parce qu'on masque la latence par le *multithreading*.

Que va-t-on faire de nos transistors ? (3/4 : gros caches L3 vs *multithreading*)

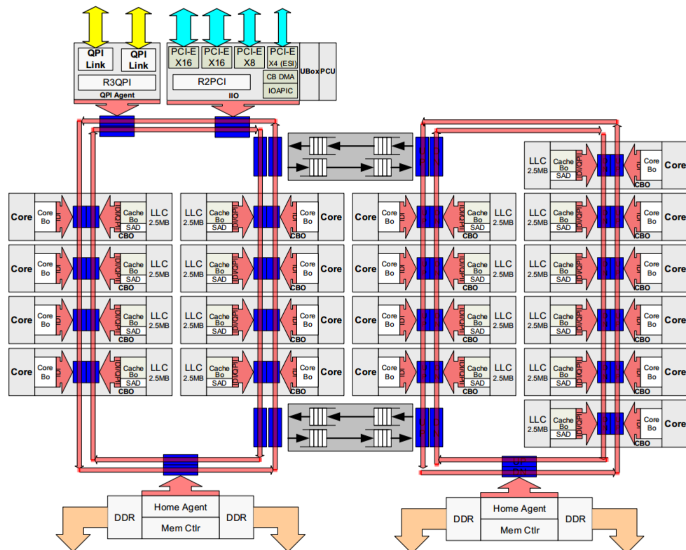
- GPU K110 = 7.1Gt, 15 SMX + 1.5MB-L2 => 1 SMX = 7000/15 = 470Mt.
- 1 SMX = 110KB-L1 (5Mt) + 64K-32b-RF (12Mt) + 192 SP (ou 64 DP) (2.5Mt/cœur-SP).
- Xeon-E5v3 = 5.6Gt, 18 cores (3.4Gt, 190Mt/cœur) + 45MB-L3 (2.2Gt).

- Pourquoi un si grand L3 dans un CPU ? Parce qu'un cœur rapide lit beaucoup de code et de données et que l'écart de vitesse avec la mémoire externe est très grand : on réduit la latence par les caches.
- Pourquoi pas de L3 dans un GPU ? Parce qu'on lit peu de code (instructions SIMD) et des données à forte localité et parce qu'on masque la latence par le *multithreading*.

- Deux voies opposées : beaucoup de petits cœurs lents ou peu de gros cœurs rapides.

GPU K110 : beaucoup de petits cœurs lents.

Xeon E5 v3 18 cores : peu de gros cœurs rapides.



Que va-t-on faire de nos transistors ? (4/4 : le *multithreading* va gagner)

- 7Gt (2015) = 2400 in-order cores + 1.5MB-L2 (GPU : 2.4G DFlops à 1Ghz) ou 20 o-o-o cores + 50MB-L3 (CPU : 0.4G DFlops à 2.5Ghz).
- 50Gt (2020) = 16K in-order cores + 10MB-L2 (GPU : 16T DFlops) ou 140 o-o-o cores + 350MB-L3 (CPU : 2.8G DFlops).
- 2Tt (2030) = 750K in-order cores + 500MB-L2 (750T DFlops).

Que va-t-on faire de nos transistors ? (4/4 : le *multithreading* va gagner)

- 7Gt (2015) = 2400 in-order cores + 1.5MB-L2 (GPU : 2.4G DFlops à 1Ghz) ou 20 o-o-o cores + 50MB-L3 (CPU : 0.4G DFlops à 2.5Ghz).
- 50Gt (2020) = 16K in-order cores + 10MB-L2 (GPU : 16T DFlops) ou 140 o-o-o cores + 350MB-L3 (CPU : 2.8G DFlops).
- 2Tt (2030) = 750K in-order cores + 500MB-L2 (750T DFlops).

- Quand on pourra exécuter un nombre illimité de *threads* sur un CPU, les gros cœurs disparaitront.

Que va-t-on faire de nos transistors ? (4/4 : le *multithreading* va gagner)

- 7Gt (2015) = 2400 in-order cores + 1.5MB-L2 (GPU : 2.4G DFlops à 1Ghz) ou 20 o-o-o cores + 50MB-L3 (CPU : 0.4G DFlops à 2.5Ghz).
- 50Gt (2020) = 16K in-order cores + 10MB-L2 (GPU : 16T DFlops) ou 140 o-o-o cores + 350MB-L3 (CPU : 2.8G DFlops).
- 2Tt (2030) = 750K in-order cores + 500MB-L2 (750T DFlops).

- Quand on pourra exécuter un nombre illimité de *threads* sur un CPU, les gros cœurs disparaitront.

- Et le GPU se fondra dans le CPU.

La parallélisation.

La séquentialité du C.

Un petit programme C.

```
1  int x;
2  void f(){
3   x=x+1;
4  }
5  void g(){
6   x=x-1;
7  }
8  void main(){
9   x=17;
10  f();
11  g();
12  printf("x=%d\n",x);
13 }
```

- L'ordre d'exécution est 9, 10, 3, 11, 6, 12.
- Pour *printf*, l'ordre 9, 11, 6, 10, 3, 12 est aussi valide.

La sémantique est donnée par l'ordre d'exécution.

- Ordre total => calcul déterministe.
- Ordre total => synchronisation des écrivains et de leurs lecteurs (un lecteur lit après l'écriture de l'écrivain).
- Ordre total => exécution séquentielle.

La parallélisation par l'OS : exemples de *pthread* et *MPI*.

Un petit programme *pthread*.

```
 1  int x;
 2  pthread_mutex_t acces_x=PTHREAD_MUTEX_INITIALIZER;
 3  void *f(void *y){
 4   pthread_mutex_lock(&acces_x);
 5   x=x+1;//x=x/2;
 6   pthread_mutex_unlock(&acces_x);
 7  }
 8  void *g(void *x){
 9   pthread_mutex_lock(&acces_x);
10   x=x-1;//x=x*2;
11   pthread_mutex_unlock(&acces_x);
12  }
13  void main(){
14   pthread_t tid1, tid2;
15   x=17;
16   pthread_create(&tid1,NULL,f,NULL);
17   pthread_create(&tid2,NULL,g,NULL);
18   pthread_join(tid1,NULL);
19   pthread_join(tid2,NULL);
20   printf("x=%d\n",x);
21  }
```

Un petit programme *pthread*.

```
1  int x;
2  pthread_mutex_t acces_x=PTHREAD_MUTEX_INITIALIZER;
3  void *f(void *y){
4   pthread_mutex_lock(&acces_x);
5   x=x+1;//x=x/2;
6   pthread_mutex_unlock(&acces_x);
7  }
8  void *g(void *x){
9   pthread_mutex_lock(&acces_x);
10  x=x-1;//x=x*2;
11  pthread_mutex_unlock(&acces_x);
12 }
13 void main(){
14  pthread_t tid1, tid2;
15  x=17;
16  pthread_create(&tid1,NULL,f,NULL);
17  pthread_create(&tid2,NULL,g,NULL);
18  pthread_join(tid1,NULL);
19  pthread_join(tid2,NULL);
20  printf("x=%d\n",x);
21 }
```

- L'ordre initialisation puis modification est garanti car 15 avant 16 et 17.
- L'ordre modification puis affichage est garanti car 20 après 18 et 19 (barrière de synchronisation *pthread_join*).
- Les modifications de x par f et g ne se mélangent pas (sérialisation par *pthread_mutex_lock* et *pthread_mutex_unlock*).

Un petit programme *MPI*.

```
1   void main(int argc, char *argv[])  {
2    int numtasks, rank, x=17, tag2_0=20, tag1_2=12;
3    MPI_Status stat;
4    MPI_Init(&argc,&argv);
5    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    if (rank==0){//main
8     MPI_Recv(&x,1,MPI_INT,2/*g*/,tag2_0,MPI_COMM_WORLD,&stat);
9     printf("x=%d\n",x);
10   } else if (rank==1){//function f
11    x=x+1;
12    MPI_Send(&x,1,MPI_INT,2/*g*/,tag1_2,MPI_COMM_WORLD);
13   }
14   else{//function g
15    MPI_Recv(&x,1,MPI_INT,1/*f*/,tag1_2,MPI_COMM_WORLD,&stat);
16    x=x-1;
17    MPI_Send(&x,1,MPI_INT,0/*main*/,tag2_0,MPI_COMM_WORLD);
18   }
19   MPI_Finalize();
20  }
```

- Les rendez-vous reconstruisent l'ordre : 2, 8, 11, 12, 15, 16, 17, 9.

La parallélisation déterministe.

Et si un programme C était exécuté en parallèle ?

```c
int f(int y){
 return (y+1);
}
int g(int y){
 return (y-1);
}
void main(){
 int x=17;
 x=f(x);
 x=g(x);
 printf("x=%d\n",x);
}
```

- Imaginons que *main*, *f* et *g* s'exécutent en parallèle.
- Comment garantir que *printf* passe après g, f passe après $x = 17$, f et g soient concurrentes et les modifications de x ne se mélangent pas ?

Et si un programme C était exécuté en parallèle ?

```c
int f(int y){
 return (y+1);
}
int g(int y){
 return (y-1);
}
void main(){
 int x=17;
 x=f(x);
 x=g(x);
 printf("x=%d\n",x);
}
```

- Imaginons que *main*, *f* et *g* s'exécutent en parallèle.
- Comment garantir que *printf* passe après g, f passe après $x = 17$, f et g soient concurrentes et les modifications de x ne se mélangent pas ?
- Réponse : en renommant et en synchronisant tout lecteur avec son écrivain.

Et si un programme C était exécuté en parallèle ?

```
1   int f(int y){
2    return (y+1);
3   }
4   int g(int y){
5    return (y-1);
6   }
7   void main(){
8    int x=17;
9    x=f(x);
10   x=g(x);
11   printf("x=%d\n",x);
12  }
```

- Imaginons que *main*, *f* et *g* s'exécutent en parallèle.
- Comment garantir que *printf* passe après *g*, *f* passe après $x = 17$, *f* et *g* soient concurrentes et les modifications de *x* ne se mélangent pas ?
- Réponse : en renommant et en synchronisant tout lecteur avec son écrivain.
- Renommer suppose qu'on dispose de l'ordre total des instructions.
- Cet ordre implique que dans la liste ordonnée des instructions à exécuter, *f* précède *g* (9 avant 10).
- Cet ordre n'implique pas que *f* s'exécute avant *g*.
- L'ordre d'exécution n'est fixé que par les dépendances producteur/consommateur ou LAE exhibées par le renommage.

Le même programme en assembleur.

```
1          .globl   main
2   f:     movq %rdi, %rax
3          addq $1, %rax
4          ret
5   g:     movq %rdi, %rax
6          subq $1, %rax
7          ret
8   main:  movq $17, %rdi
9          call f
10         movq %rax, %rdi
11         call g
12         movq $.LC0, %rdi
13         movq %rax, %rsi
14         movq $0, %rax
15         call printf
16         ret
17  .LC0:
18         .string "x=%d\n"
```

Le même programme après parallélisation.

```
1          .globl   main
2   f:     movq %rdi, %rax
3          addq $1, %rax
4          endfork
5   g:     movq %rdi, %rax
6          subq $1, %rax
7          endfork
8   main:  movq $17, %rdi
9          fork f
10         movq %rax, %rdi
11         fork g
12         movq $.LC0, %rdi
13         movq %rax, %rsi
14         movq $0, %rax
15         fork printf
16         endfork
17  .LC0:
18         .string "x=%d\n"
```

Fork et endfork.

```
1          .globl   main
2   f:     movq %rdi, %rax
3          addq $1, %rax
4          endfork
5   g:     movq %rdi, %rax
6          subq $1, %rax
7          endfork
8   main:  movq $17, %rdi
9          fork f
10         movq %rax, %rdi
11         fork g
12         movq $.LC0, %rdi
13         movq %rax, %rsi
14         movq $0, %rax
15         fork printf
16         endfork
17  .LC0:
18         .string "x=%d\n"
```

- *fork* remplace *call* en lignes 9, 11 et 15.
- *fork f* exécute en parallèle (2,3,4) et (10,11).
- *fork g* exécute en parallèle (5,6,7) et (12-15).
- *fork printf* exécute en parallèle 16 et *printf*.
- Quatre flots concurrents : (8,9,2-4) | (10,11,5-7) | (12-15,...) | (16).

Le même programme après renommage.

```
1          .globl   main
2   f:     rax0 = rdi0
3          rax1 = rax0+1
4          endfork
5   g:     rax2 = rdi1
6          rax3 = rax2-1
7          endfork
8   main:  rdi0 = 17
9          fork f
10         rdi1 = rax1
11         fork g
12         rdi2 = $.LC0
13         rsi0 = rax3
14         rax4 = 0
15         fork printf
16         endfork
17  .LC0:
18         .string "x=%d\n"
```

- Imaginons que toutes les destinations soient rendues uniques par renommage matériel.

Le même programme mélangé.

```
1          rax4  = 0
2          rsi0  = rax3
3          rdi2  = $.LC0
4          rdi1  = rax1
5          rdi0  = 17
6          rax3  = rax2-1
7          rax2  = rdi1
8          rax1  = rax0+1
9          rax0  = rdi0
10 .LC0:
11         .string "x=%d\n"
```

- Malgré le mélange, la sémantique ne change pas (mais un lecteur doit attendre son écrivain).

Les instructions indépendantes sont parallélisées et les autres sont synchronisées.

- L'exécution des instructions 2-16 peut se faire dans un ordre quelconque si chaque lecteur est synchronisé avec son écrivain.

Les instructions indépendantes sont parallélisées et les autres sont synchronisées.

- L'exécution des instructions 2-16 peut se faire dans un ordre quelconque si chaque lecteur est synchronisé avec son écrivain.
- La lecture des instructions crée la trace totalement ordonnée 8,9,2,3,4,10,11,5,6,7,12,13,14,15,*printf*,16.

Les instructions indépendantes sont parallélisées et les autres sont synchronisées.

- L'exécution des instructions 2-16 peut se faire dans un ordre quelconque si chaque lecteur est synchronisé avec son écrivain.
- La lecture des instructions crée la trace totalement ordonnée 8,9,2,3,4,10,11,5,6,7,12,13,14,15,*printf*,16.
- Quand la trace est lue et renommée, on peut tout exécuter en parallèle (sans se soucier de l'ordre induit par la lecture).
- Un lecteur ne peut lire qu'après que la variable qu'il lit soit écrite.

Les instructions indépendantes sont parallélisées et les autres sont synchronisées.

- L'exécution des instructions 2-16 peut se faire dans un ordre quelconque si chaque lecteur est synchronisé avec son écrivain.
- La lecture des instructions crée la trace totalement ordonnée 8,9,2,3,4,10,11,5,6,7,12,13,14,15,*printf*,16.
- Quand la trace est lue et renommée, on peut tout exécuter en parallèle (sans se soucier de l'ordre induit par la lecture).
- Un lecteur ne peut lire qu'après que la variable qu'il lit soit écrite.
- Ordre partiel de terminaison des exécutions : (4 | 7 | 8 | 9 | 11 | 12 | 14 | 15 | 16) puis 2 (qui dépend de 8) puis 3 (qui dépend de 2) puis 10 (qui dépend de 3) puis 5 (qui dépend de 10) puis 6 (qui dépend de 5) puis 13 (qui dépend de 6).

Un autre exemple : asynchronisme par interruption.

```
int i=0; float t[4]; typedef struct {int c;} ST;
pthread_mutex_t acces_i=PTHREAD_MUTEX_INITIALIZER;
float capteur_handler(int c){//capteurs interrupt handler
 float f; phys_read(capteur[c],&f,sizeof(float)); return f;
}
void *get_c(void *st){
 float f = capteur_read((ST *)st->c);//sleeps until interrupt returns
 pthread_mutex_lock(&acces_i); t[i++]=f;//t is filled in interrupt order
 pthread_mutex_unlock(&acces_i); pthread_exit(NULL);
}
void work_on(float f){...}
main(){
 pthread_t tid[4]; ST s[4]={{0}, {1}, {2}, {3}};
 int count=0, c;
 for(c=0;c<4;c++) pthread_create(&tid[c],NULL,get_c,(void *)&s[c]);
 while(count<4){
  for(c=0;c<4;c++)
   if (tid[c] && pthread_tryjoin_np(tid[c],NULL)>=0)
      {work_on(t[count]); count++; tid[c]=0;}
 }
}
```

Un autre exemple : asynchronisme par interruption.

```
int i=0; float t[4]; typedef struct {int c;} ST;
pthread_mutex_t acces_i=PTHREAD_MUTEX_INITIALIZER;
float capteur_handler(int c){//capteurs interrupt handler
 float f; phys_read(capteur[c],&f,sizeof(float)); return f;
}
void *get_c(void *st){
 float f = capteur_read((ST *)st->c);//sleeps until interrupt returns
 pthread_mutex_lock(&acces_i); t[i++]=f;//t is filled in interrupt order
 pthread_mutex_unlock(&acces_i); pthread_exit(NULL);
}
void work_on(float f){...}
main(){
 pthread_t tid[4]; ST s[4]={{0}, {1}, {2}, {3}};
 int count=0, c;
 for(c=0;c<4;c++) pthread_create(&tid[c],NULL,get_c,(void *)&s[c]);
 while(count<4){
  for(c=0;c<4;c++)
   if (tid[c] && pthread_tryjoin_np(tid[c],NULL)>=0)
      {work_on(t[count]); count++; tid[c]=0;}
 }
}
```

- L'ordre dans *t* est celui des interruptions.
- La fonction *work_on* est appelée 4 fois, dans l'ordre des interruptions.
- Le calcul est asynchrone (on peut paralléliser les *work_on* : *threads*).
- Le calcul est non déterministe (p.ex. *work_on* somme les flottants $s+ = t[count]$: somme en ordre variable).

Version C du programme.

```
int i=0; float t[4];
float capteur_handler(int c){//capteurs interrupt handler
 float f; phys_read(capteur[c],&f,sizeof(float)); return f;
}
void get_c(int c){
 float f = capteur_read(c);//sleeps until interrupt returns
 t[i++]=f;
}
void work_on(float f){...}
main(){
 int c;
 for(c=0;c<4;c++) get_c(c);
 for(c=0;c<4;c++) work_on(t[c]);
}
```

- L'ordre dans *t* est celui des 4 appels à *get_c*.
- Les interruptions sont asynchrones.
- Le calcul n'est pas asynchrone. Il est séquentiel.
- Le calcul est déterministe (ordre fixe des *work_on*).

Parallélisation du programme C.

```
int i=0; float t[4];
float capteur_handler(int c){
 float f; phys_read(capteur[c],&f,sizeof(float)); return f;
}
void get_c(int c){
 float f = capteur_read(c);//attente de l'entrée
 t[i++]=f;
}
void work_on(float f){...}
main(){
 int c;
 for(c=0;c<4;c++) get_c(c);
 for(c=0;c<4;c++) work_on(t[c]);
}
```

Parallélisation du programme C.

```
int i=0; float t[4];
float capteur_handler(int c){
 float f; phys_read(capteur[c],&f,sizeof(float)); return f;
}
void get_c(int c){
 float f = capteur_read(c);//attente de l'entrée
 t[i++]=f;
}
void work_on(float f){...}
main(){
 int c;
 for(c=0;c<4;c++) get_c(c);
 for(c=0;c<4;c++) work_on(t[c]);
}
```

- On construit une trace par lecture en parallèle des instructions à exécuter.
- Ordre total incluant le code de *capteur_handler* (*get_c* contient *capteur_read* qui contient *capteur_handler* qui contient *phys_read*).
- On renomme la trace en parallèle.
- On exécute les instructions renommées en ordre partiel des dépendances LAE.

Lecture, renommage et exécution parallèles des instructions du programme C.

```
//ordre de la trace renommée
i0=0      ;  c0=0;
get_c(c0);  f0=?;  t[i0]=f0;
i1=i0+1   ;  c1=1;
get_c(c1);  f1=?;  t[i1]=f1;
i2=i1+1   ;  c2=2;
get_c(c2);  f2=?;  t[i2]=f2;
i3=i2+1   ;  c3=3;
get_c(c3);  f3=?;  t[i3]=f3;
i4=i3+1   ;  c4=0;  work_on(t[c4]);
             c5=1;  work_on(t[c5]);
             c6=2;  work_on(t[c6]);
             c7=3;  work_on(t[c7]);
```

Lecture, renommage et exécution parallèles des instructions du programme C.

```
//ordre de la trace renommée
i0=0       ; c0=0;
get_c(c0); f0=?; t[i0]=f0;
i1=i0+1    ; c1=1;
get_c(c1); f1=?; t[i1]=f1;
i2=i1+1    ; c2=2;
get_c(c2); f2=?; t[i2]=f2;
i3=i2+1    ; c3=3;
get_c(c3); f3=?; t[i3]=f3;
i4=i3+1    ; c4=0; work_on(t[c4]);
             c5=1; work_on(t[c5]);
             c6=2; work_on(t[c6]);
             c7=3; work_on(t[c7]);
```

```
//ordre d'exécution
i0=0        c0=0 c1=1 c2=2 c3=3
            c4=0 c5=1 c6=2 c7=3;
i1=i0+1     get_c(c0) get_c(c1)
            get_c(c2) get_c(c3);
i2=i1+1;    i3=i2+1;  i4=i3+1;
//attente d'une entrée
f3=?;       t[i3]=f3; work_on(t[c7])
f1=?;       t[i1]=f1; work_on(t[c5])
f0=?;       t[i0]=f0; work_on(t[c4])
f2=?;       t[i2]=f2; work_on(t[c6])
```

Lecture, renommage et exécution parallèles des instructions du programme C.

```
//ordre de la trace renommée
i0=0       ; c0=0;
get_c(c0); f0=?;  t[i0]=f0;
i1=i0+1   ; c1=1;
get_c(c1); f1=?;  t[i1]=f1;
i2=i1+1   ; c2=2;
get_c(c2); f2=?;  t[i2]=f2;
i3=i2+1   ; c3=3;
get_c(c3); f3=?;  t[i3]=f3;
i4=i3+1   ; c4=0; work_on(t[c4]);
             c5=1; work_on(t[c5]);
             c6=2; work_on(t[c6]);
             c7=3; work_on(t[c7]);
```

```
//ordre d'exécution
i0=0       c0=0  c1=1  c2=2  c3=3
           c4=0  c5=1  c6=2  c7=3;
i1=i0+1    get_c(c0) get_c(c1)
           get_c(c2) get_c(c3);
i2=i1+1;   i3=i2+1;  i4=i3+1;
//attente d'une entrée
f3=?;      t[i3]=f3; work_on(t[c7])
f1=?;      t[i1]=f1; work_on(t[c5])
f0=?;      t[i0]=f0; work_on(t[c4])
f2=?;      t[i2]=f2; work_on(t[c6])
```

- Ordre total de la trace renommée (= ordre total du code séquentiel).
- Renommage de *capteur_handler*(c)
  => l'ordre dans t est celui des 4 appels à *get_c*. ($t[i0] = f0$ ... $t[i3] = f3$).
- Exécution des *work_on* dans l'ordre des entrées.
- Le calcul est asynchrone. Les *work_on* sont parallèles.
- Le calcul est déterministe (ordre fixe des sommes car *dataflow* : $s0 = 0.0$, $s4 = s3 + t[i3]$, $s2 = s1 + t[i1]$, $s1 = s0 + t[i0]$, $s3 = s2 + t[i2]$).

Deux ordres : ordre total de lecture/renommage, ordre partiel d'exécution.

- Au démarrage, un processus initial.

Deux ordres : ordre total de lecture/renommage, ordre partiel d'exécution.

- Au démarrage, un processus initial.
- Un premier *fork* : le second processus se place après le premier.
- Ils sont exécutés en parallèle mais en respectant les dépendances LAE.

Deux ordres : ordre total de lecture/renommage, ordre partiel d'exécution.

- Au démarrage, un processus initial.
- Un premier *fork* : le second processus se place après le premier.
- Ils sont exécutés en parallèle mais en respectant les dépendances LAE.
- Les événements asynchrones (p.ex. la connexion d'un utilisateur) prennent d'abord une place dans l'ordre total : création d'un processus de connexion lié à un terminal qui attend le clavier.
- La saisie au clavier remplit un flux qui déclenche l'exécution de la connexion.

Deux ordres : ordre total de lecture/renommage, ordre partiel d'exécution.

- Au démarrage, un processus initial.
- Un premier *fork* : le second processus se place après le premier.
- Ils sont exécutés en parallèle mais en respectant les dépendances LAE.
- Les événements asynchrones (p.ex. la connexion d'un utilisateur) prennent d'abord une place dans l'ordre total : création d'un processus de connexion lié à un terminal qui attend le clavier.
- La saisie au clavier remplit un flux qui déclenche l'exécution de la connexion.
- L'ordre total des lectures/renommages et l'ordre partiel des exécutions donne une exécution parallèle déterministe.

Les vertus du déterminisme.

- Les machines actuelles lancent des processus et des *threads* indépendants, ce qui fait perdre le déterminisme.
- Elles tentent de retrouver ce déterminisme en empêchant des ordres d'exécution inacceptables par la synchronisation : sémaphores, mutex, barrières, rendez-vous, moniteurs ...
- Ca peut marcher, mais c'est très complexe à vérifier (le plus souvent, pas de preuve) et très malaisé à mettre au point (non reproductibilité des erreurs).

Les vertus du déterminisme.

- Les machines actuelles lancent des processus et des *threads* indépendants, ce qui fait perdre le déterminisme.
- Elles tentent de retrouver ce déterminisme en empêchant des ordres d'exécution inacceptables par la synchronisation : sémaphores, mutex, barrières, rendez-vous, moniteurs ...
- Ca peut marcher, mais c'est très complexe à vérifier (le plus souvent, pas de preuve) et très malaisé à mettre au point (non reproductibilité des erreurs).
- Le déterminisme permet la preuve et la reproductibilité (des résultats comme des erreurs).

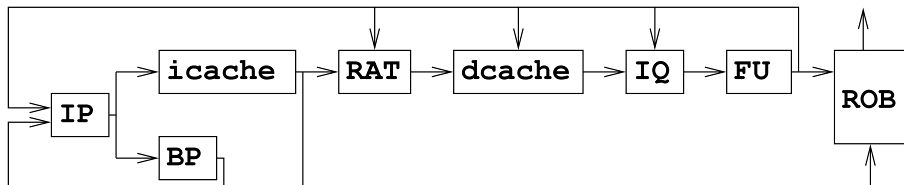
La parallélisation par le matériel.

Un cœur à exécution spéculative (cœur actuel).

Un cœur d'aujourd'hui.

- Algorithme de Tomasulo (1967).
- Renommage et exécution en désordre.
- Prédiction des sauts et exécution spéculative.

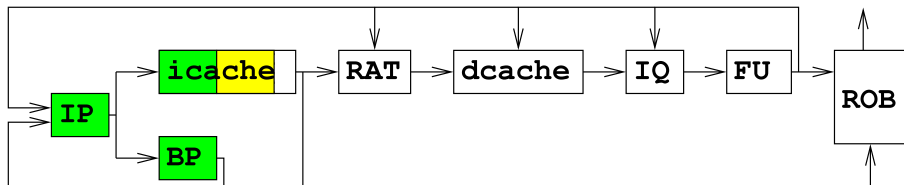
Pipeline du cœur.



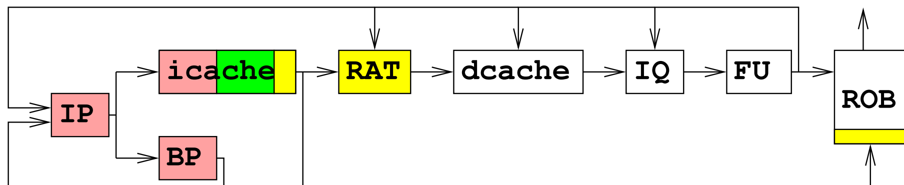
La lecture des instructions et la prédiction.

IP
icache
BP
RAT
dcache
IQ
FU
ROB

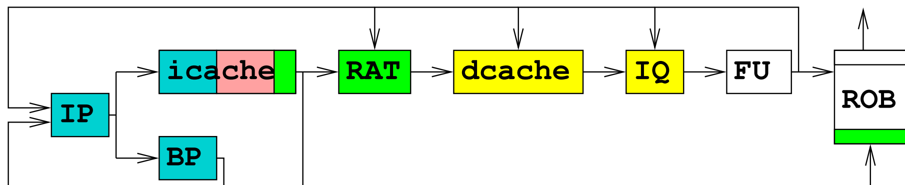
La lecture (cycle 2 : on suit le carré jaune).



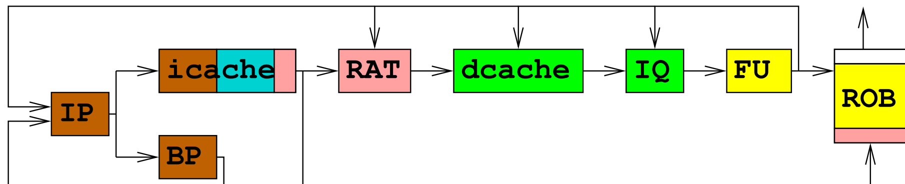
La lecture (cycle 3), le décodage et le renommage.



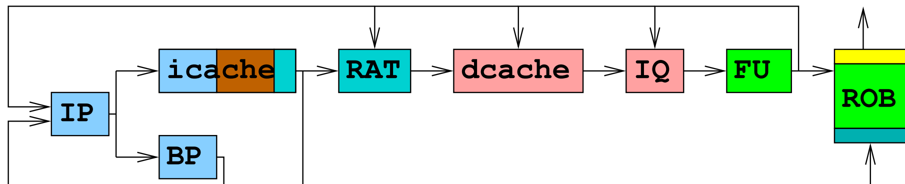
La lecture des sources prêtes.



L'exécution des instructions prêtes.

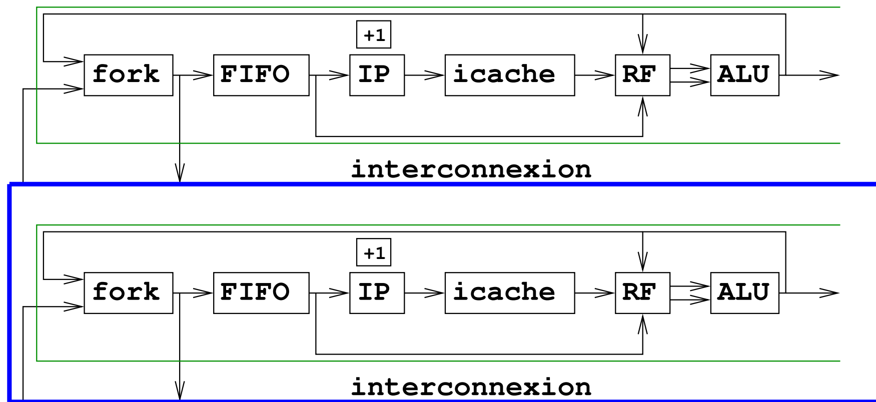


Le retrait en ordre.



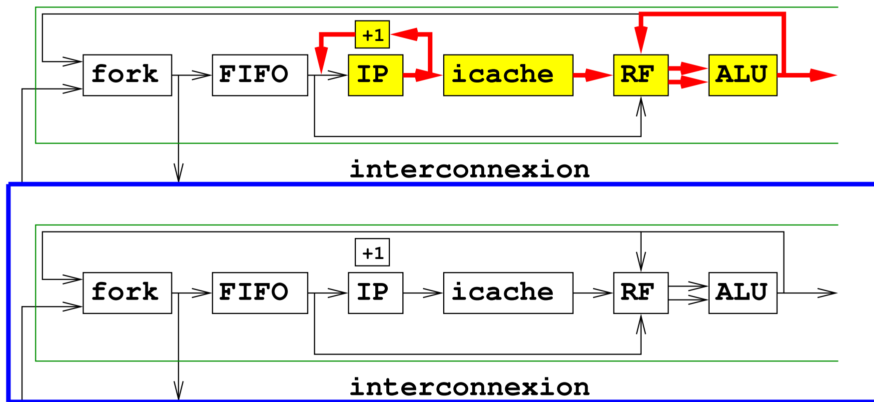
Un cœur parallélisant (cœur du futur).

La lecture et le décodage des instructions.

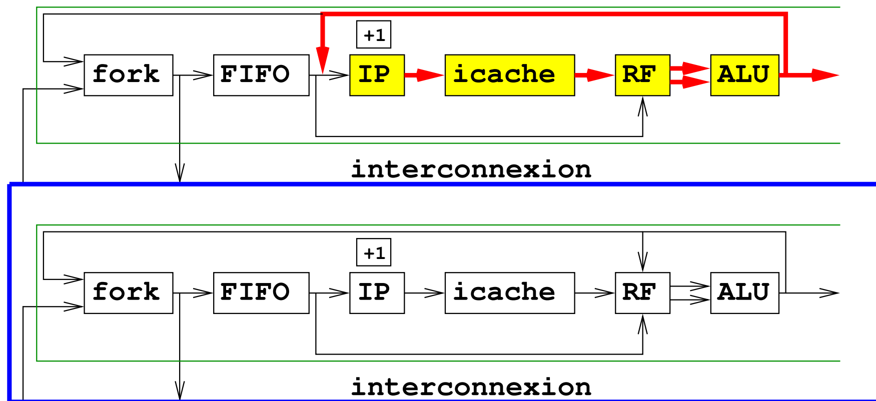


Le renommage et l'exécution.

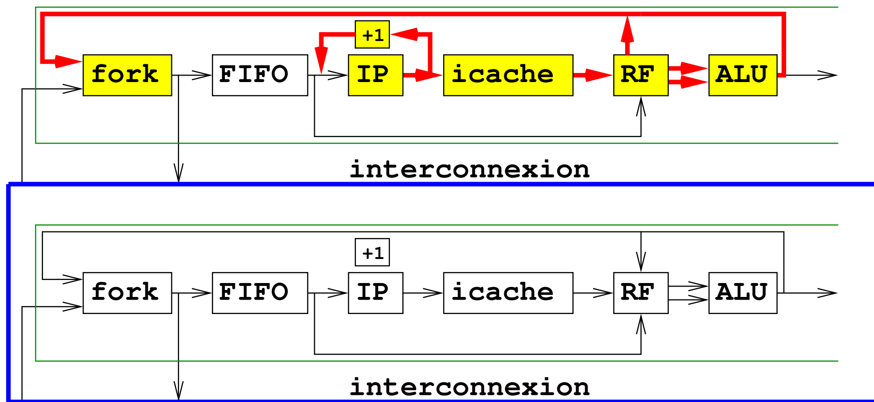
Le cycle va de l'instruction au résultat du calcul.



Un saut calculé.

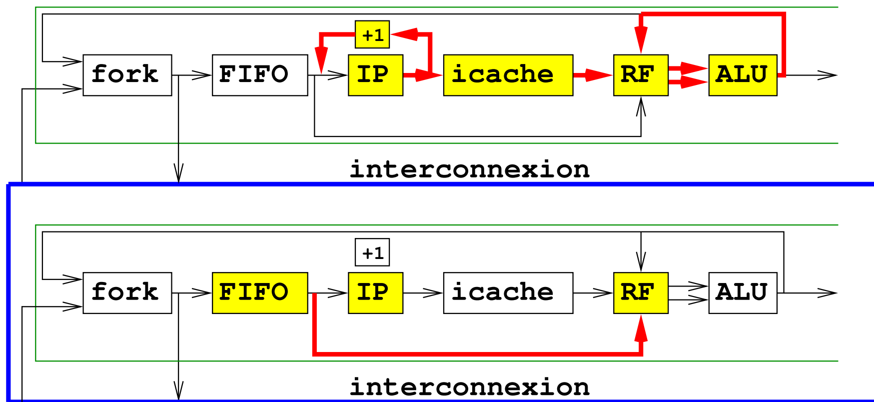


Instruction *fork* : copie de registres.

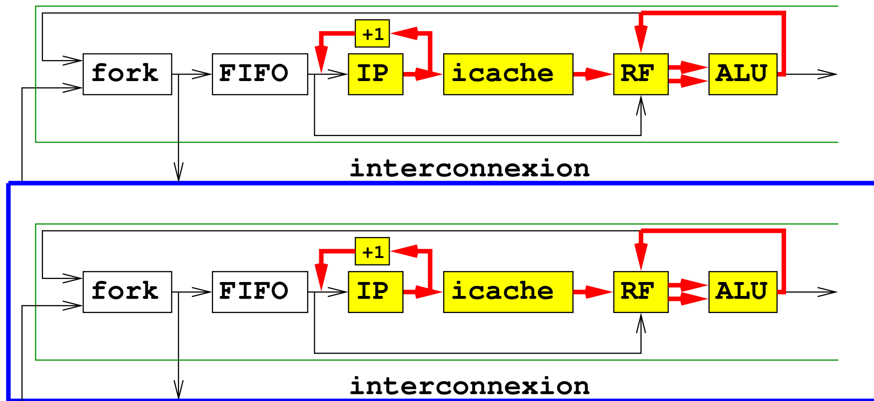


Instruction *fork* : transmission IP et registres.

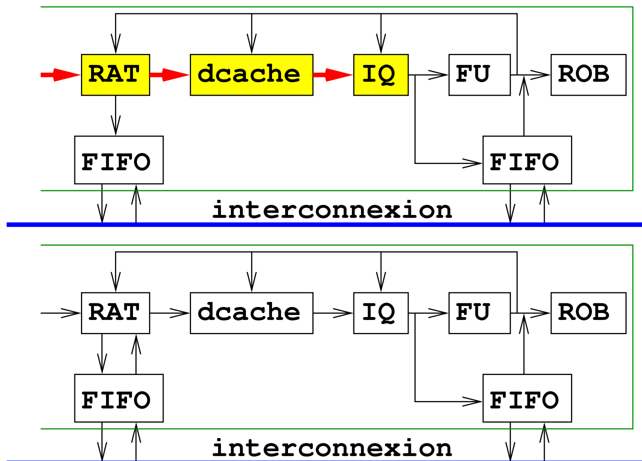
Instruction *fork* : initialisation IP et registres.



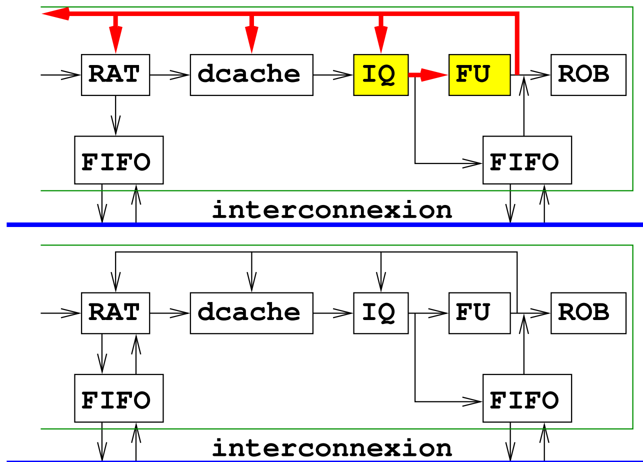
Instruction *fork* : deux cœurs actifs.



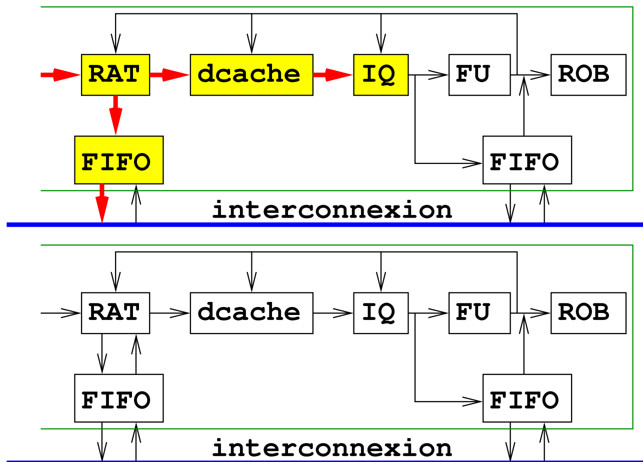
Renommage en ordre des registres et de la mémoire.



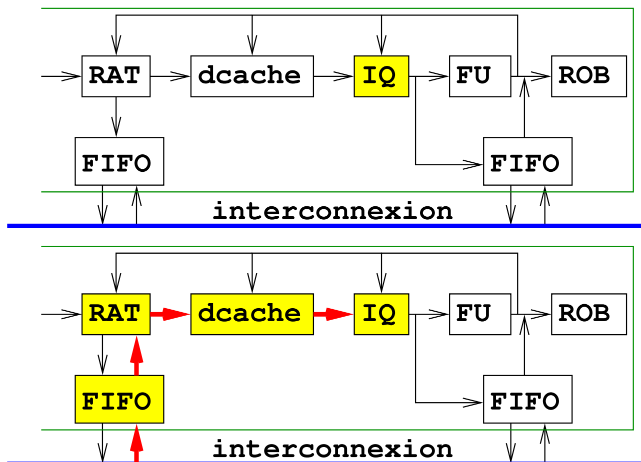
Exécution en désordre des instructions renommées.



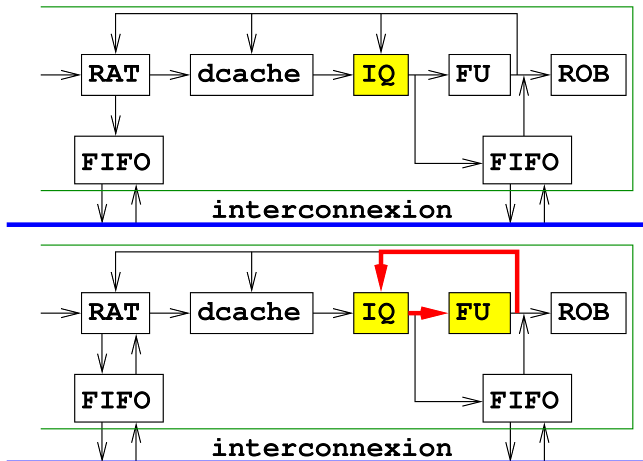
Renommage d'une source produite par une autre section.



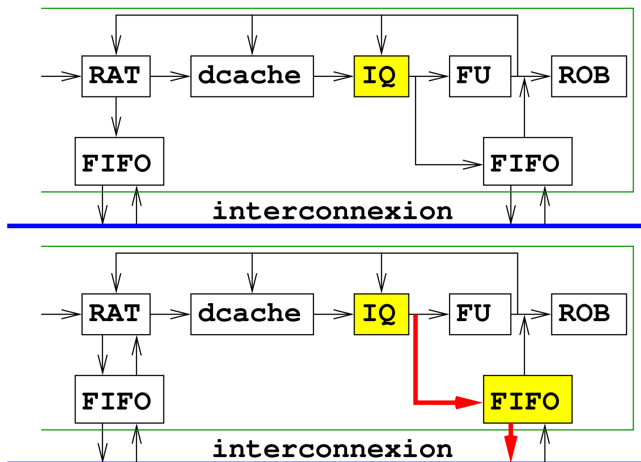
Renommage par le producteur.



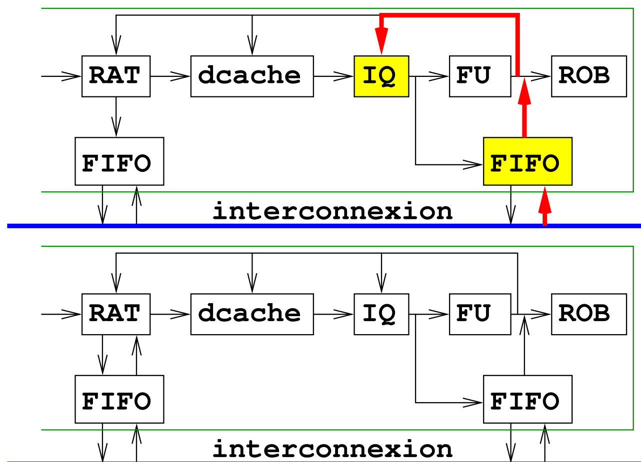
Calcul de la valeur à exporter et diffusion locale.



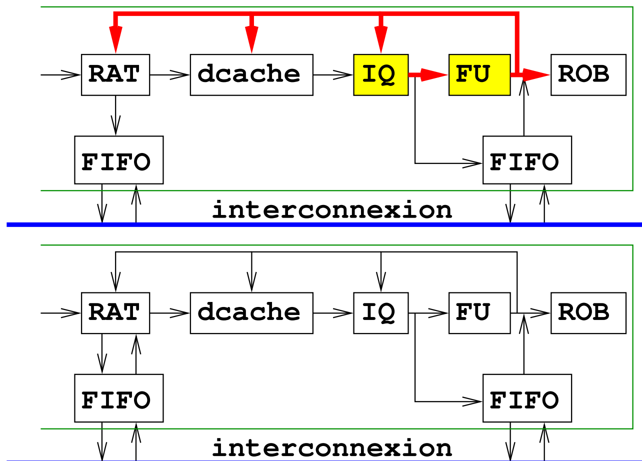
Export.



Réception et diffusion locale.



Calcul à partir de la source importée.



La parallélisation matérielle sur un exemple.

Exemple : une réduction de somme.

```
long rDS(long t[], unsigned int n){
 if (n==1) return t[0];
 if (n==2) return t[0]+t[1];
 return rDS(t,n/2) + rDS(&(t[n/2]),n-n/2);
}
```

Exemple : une réduction de somme.

```c
long rDS(long t[], unsigned int n){
  if (n==1) return t[0];
  if (n==2) return t[0]+t[1];
  return rDS(t,n/2) + rDS(&(t[n/2]),n-n/2);
}
```

```
rDS:    cmpq     $2, %rsi              ;if (n>2) goto .L1
        ja       .L1
        movq     (%rdi), %rax          ;rax = t[0]
        jb       .L2                   ;if (n<2) goto .L2
        addq     8(%rdi), %rax         ;rax = t[0] + t[1]
.L2:    endfork                        ;return (rax)
.L1:    subq     $8, %rsp              ;alloc (temp)
        movq     %rsi, %rbp            ;rbp = n
        shrq     %rsi                  ;n = n/2
        fork     rDS                   ;rax = rDS(t, n/2)
        movq     %rax, 0(%rsp)         ;temp = t[0] + ... + t[n/2-1]
        leaq     (%rdi, %rsi, 8), %rdi ;t = t + n/2*8 = &(t[n/2])
        subq     %rsi, %rbp            ;rbp = n - n/2
        movq     %rbp, %rsi            ;n = n - n/2
        fork     rDS                   ;rax = rDS(&(t[n/2]), n-n/2)
        addq     0(%rsp), %rax         ;rax = t[0]   + ... + t[n/2-1]
                                       ;    + t[n/2] + ... + t[n-1]
        addq     $8, %rsp              ;free (temp)
        endfork                        ;return (rax)
```

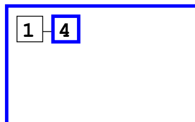
La lecture du code en parallèle.

La lecture du code : calcul de saut.

```
rDS:  cmpq $2, %rsi                      1
      ja .L1

      movq (%rdi), %rax                  2
      jb .L2

      addq 8(%rdi), %rax                 3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp                    4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                    5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                      6
      endfork
```

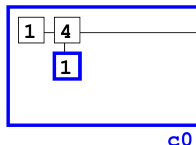
La lecture du code : préparation de *fork*.

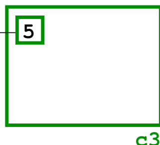
```
rDS:  cmpq $2, %rsi                      1
      ja .L1

      movq (%rdi), %rax                  2
      jb .L2

      addq 8(%rdi), %rax                 3
.L2:  endfork

.L1:  subq $8, %rsp                      4
      movq %rsi, %rbp
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)                 5
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax                 6
      addq $8, %rsp
      endfork
```



La lecture du code : *fork* avec copie de registres.

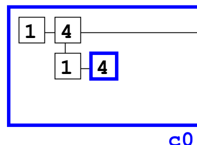
```
rDS:  cmpq $2, %rsi                  1
      ja .L1
      movq (%rdi), %rax              2
      jb .L2
      addq 8(%rdi), %rax             3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                      4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                  6
      endfork
```

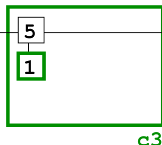




La lecture du code : fin de section.

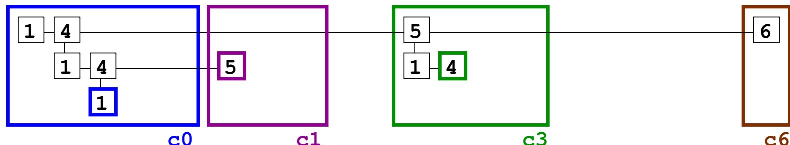
```
rDS:  cmpq $2, %rsi                 1
      ja .L1

      movq (%rdi), %rax             2
      jb .L2

      addq 8(%rdi), %rax            3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp               4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp               5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                 6
      endfork
```



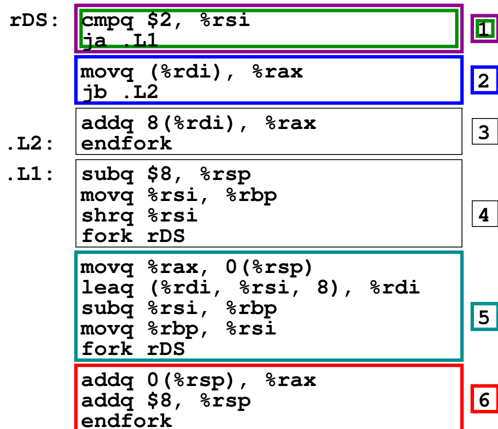


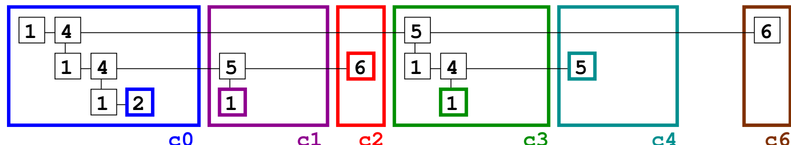
La lecture du code : deux *fork* en parallèle.

```
rDS:  cmpq $2, %rsi                 1
      ja .L1
      movq (%rdi), %rax             2
      jb .L2
      addq 8(%rdi), %rax            3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp               4
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp               5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                 6
      endfork
```



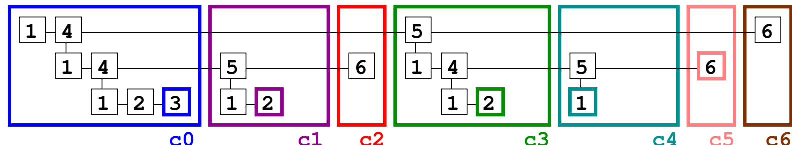
La lecture du code : 5 sections lues en parallèle.





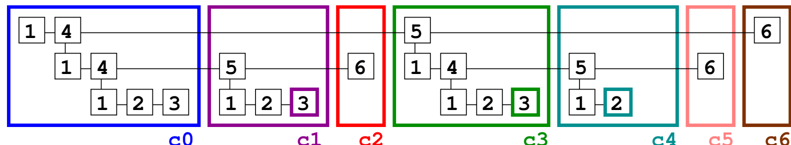
La lecture du code : le même code lu 2 fois.

```
rDS:  cmpq $2, %rsi              1
      ja .L1
      movq (%rdi), %rax          2
      jb .L2
      addq 8(%rdi), %rax         3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                  4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp            5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp              6
      endfork
```



La lecture du code : épilogue.

```
rDS:  cmpq $2, %rsi                  1
      ja .L1
      movq (%rdi), %rax              2
      jb .L2
      addq 8(%rdi), %rax             3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                      4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                  6
      endfork
```



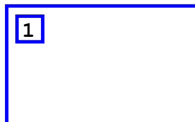
La lecture du code : fin de lecture.

```
rDS:  cmpq $2, %rsi                  [1]
      ja .L1
      movq (%rdi), %rax              [2]
      jb .L2
      addq 8(%rdi), %rax             [3]
.L2:  endfork
.L1:  subq $8, %rsp                  [4]
      movq %rsi, %rbp
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)             [5]
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax             [6]
      addq $8, %rsp
      endfork
```

Le renommage et l'exécution parallèles.

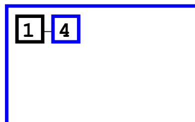
Renommage et exécution : $rsi = 8$, $rdi = t$, $rsp = SP$

```
rDS:  cmpq $2, %rsi                 1
      ja .L1

      movq (%rdi), %rax             2
      jb .L2

      addq 8(%rdi), %rax            3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp               4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp               5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                 6
      endfork
```



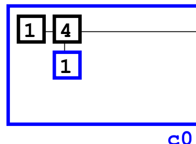
Renommage et exécution : exécution sans renommage

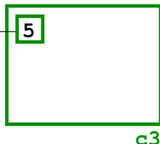
```
rDS:  cmpq $2, %rsi                    1
      ja .L1

      movq (%rdi), %rax                2
      jb .L2

      addq 8(%rdi), %rax               3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp                  4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                  5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                    6
      endfork
```



Renommage et exécution : c3 reçoit *rbp* = 8 et *rsi* = 4

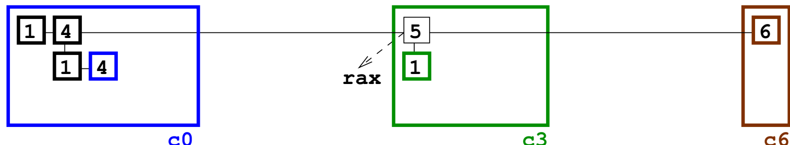
```
rDS:  cmpq $2, %rsi                  1
      ja .L1

      movq (%rdi), %rax              2
      jb .L2

      addq 8(%rdi), %rax             3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp                4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                  6
      endfork
```





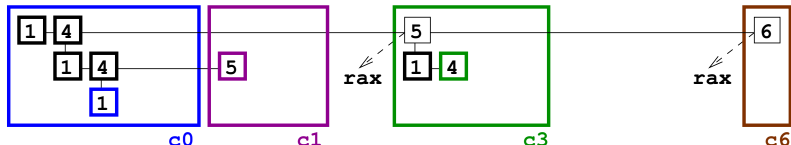
Renommage et exécution : c3 renomme *rax* et 0(*rsp*)

```
rDS:  cmpq $2, %rsi                 1
      ja .L1
      movq (%rdi), %rax             2
      jb .L2
      addq 8(%rdi), %rax            3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp               4
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp               5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                 6
      endfork
```

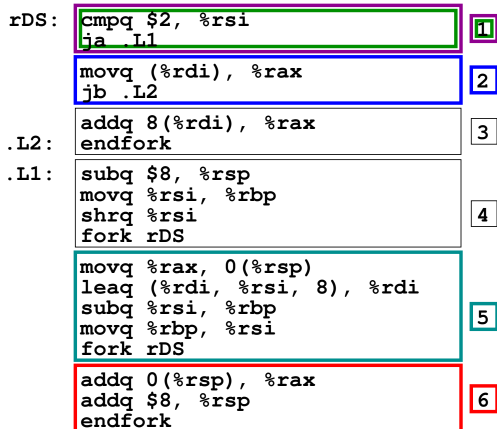


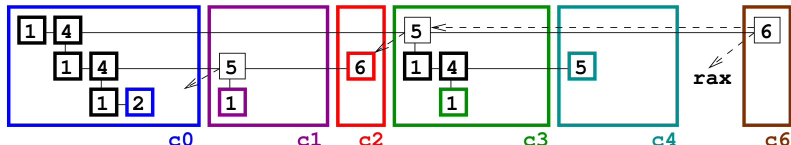
Renommage et exécution : c1 reçoit $rbp = 4$ et $rsi = 2$

```
rDS:  cmpq $2, %rsi                     1
      ja .L1

      movq (%rdi), %rax                 2
      jb .L2

      addq 8(%rdi), %rax                3
.L2:  endfork

.L1:  subq $8, %rsp                     4
      movq %rsi, %rbp
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)                5
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax                6
      addq $8, %rsp
      endfork
```

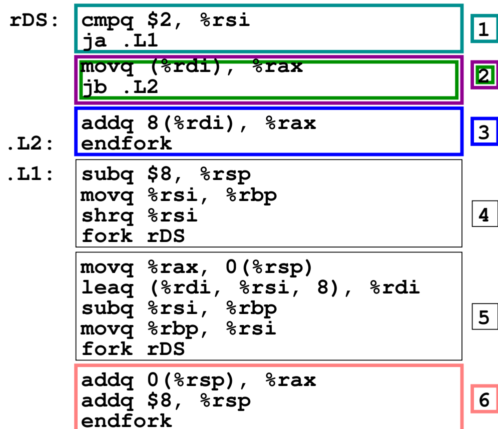


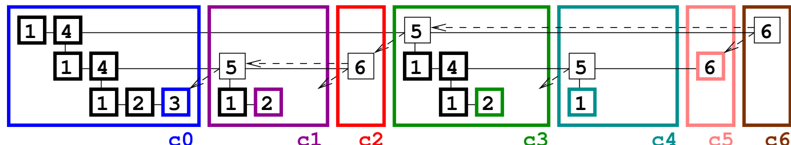
Renommage et exécution : c2 doit envoyer *rax* à c3



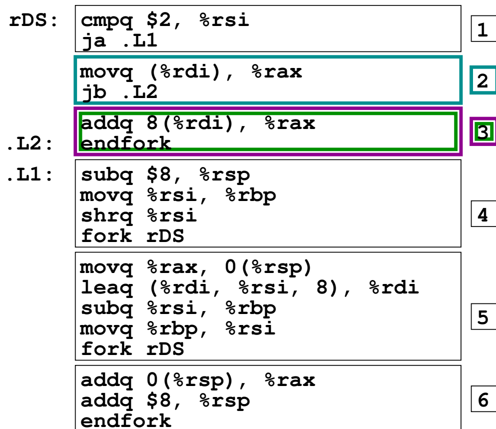


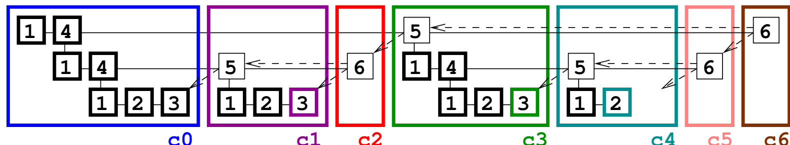
Renommage et exécution : c0 doit envoyer *rax* à c1





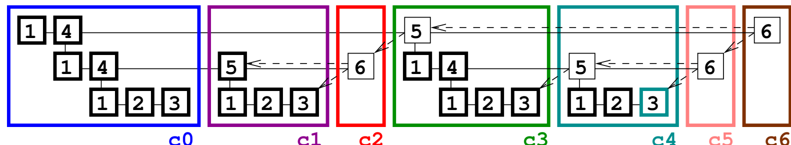
Renommage et exécution : c0 calcule *rax*





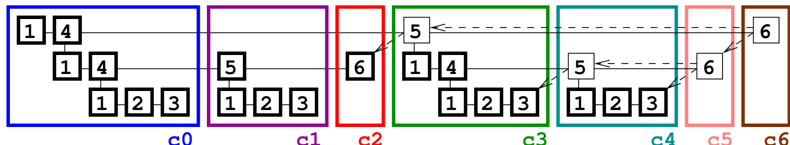
Renommage et exécution : c0 envoie *rax* à c1

```
rDS:  cmpq $2, %rsi                    1
      ja .L1
      movq (%rdi), %rax                2
      jb .L2
      addq 8(%rdi), %rax               3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                        4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                  5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                    6
      endfork
```



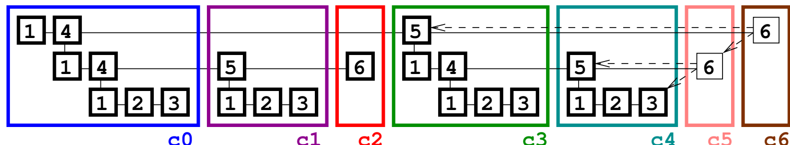
Renommage et exécution : c1 envoie *rax* et 0(*rsp*) à c2

```
rDS:  cmpq $2, %rsi                      1
      ja .L1
      movq (%rdi), %rax                  2
      jb .L2
      addq 8(%rdi), %rax                 3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                          4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                    5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                      6
      endfork
```



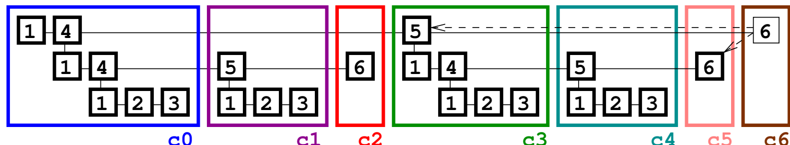
Renommage et exécution : c2/3 envoient *rax* à c3/4

```
rDS:  cmpq $2, %rsi                  1
      ja  .L1
      movq (%rdi), %rax              2
      jb  .L2
      addq 8(%rdi), %rax             3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                      4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                  6
      endfork
```



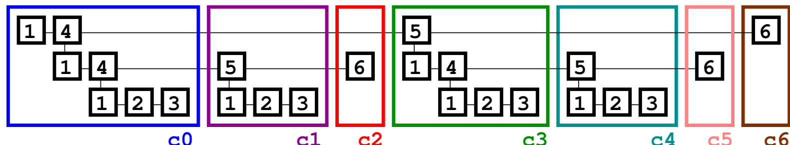
Renommage et exécution : c4 envoie *rax* et 0(*rsp*) à c5

```
rDS:  cmpq $2, %rsi                              1
      ja  .L1
      movq (%rdi), %rax                          2
      jb  .L2
      addq 8(%rdi), %rax                         3
.L2:  endfork
.L1:  subq $8, %rsp                              4
      movq %rsi, %rbp
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)                         5
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax                         6
      addq $8, %rsp
      endfork
```



Renommage et exécution : c3/5 envoient 0(*rsp*)/*rax* à c6

```
rDS:  cmpq $2, %rsi                  1
      ja .L1
      movq (%rdi), %rax              2
      jb .L2
      addq 8(%rdi), %rax             3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                      4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                  6
      endfork
```



Le retrait parallèle.

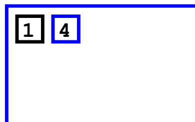
Retrait : aucun

```
rDS:  cmpq $2, %rsi                      1
      ja .L1

      movq (%rdi), %rax                  2
      jb .L2

      addq 8(%rdi), %rax                 3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp                    4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                    5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                      6
      endfork
```

c0

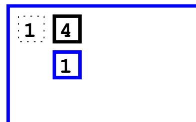
Retrait : aucun

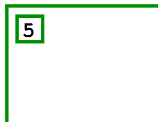
```
rDS:  cmpq $2, %rsi                 1
      ja .L1

      movq (%rdi), %rax             2
      jb .L2

      addq 8(%rdi), %rax            3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp               4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp               5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                 6
      endfork
```



Retrait : c0-bloc 1, aucun export

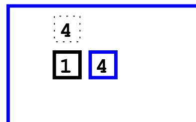
```
rDS:  cmpq $2, %rsi                          1
      ja .L1

      movq (%rdi), %rax                      2
      jb .L2

      addq 8(%rdi), %rax                     3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp                        4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                        5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                          6
      endfork
```



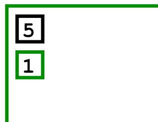


Retrait : c0-bloc 4, aucun export

```
rDS:  cmpq $2, %rsi                          [1]
      ja .L1
      movq (%rdi), %rax                      [2]
      jb .L2
      addq 8(%rdi), %rax                     [3]
.L2:  endfork
.L1:  subq $8, %rsp                          [4]
      movq %rsi, %rbp
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)                     [5]
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax                     [6]
      addq $8, %rsp
      endfork
```



c0

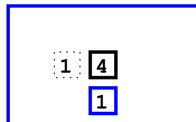


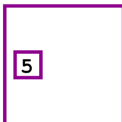
c3

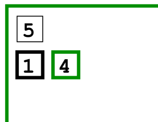
c6

Retrait : c0-bloc 1, aucun export

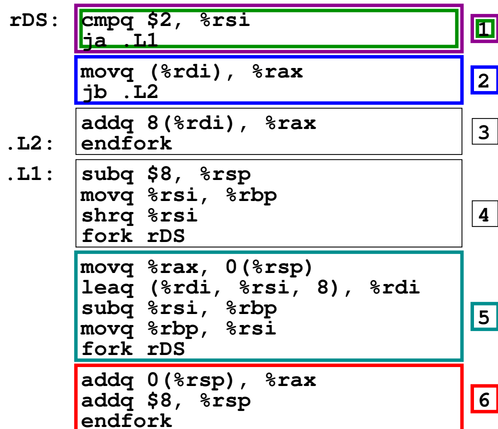
```
rDS:  cmpq $2, %rsi                  1
      ja .L1
      movq (%rdi), %rax              2
      jb .L2
      addq 8(%rdi), %rax             3
.L2:  endfork
.L1:  subq $8, %rsp                  4
      movq %rsi, %rbp
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)             5
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax             6
      addq $8, %rsp
      endfork
```

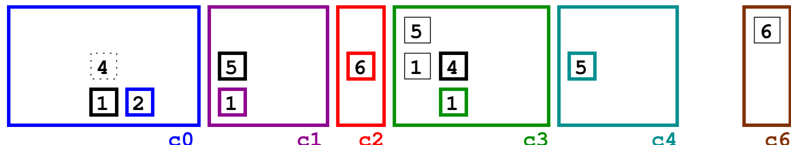




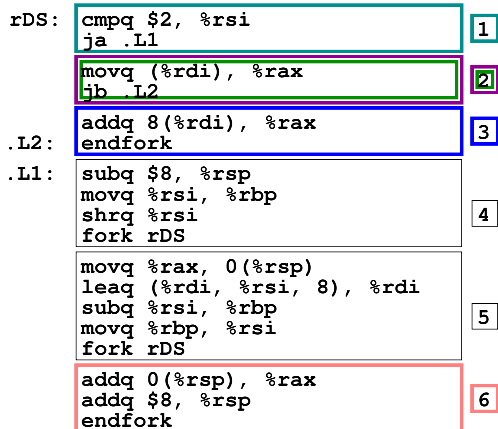


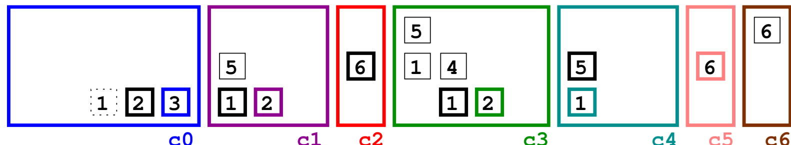
Retrait : c0-bloc 4, aucun export





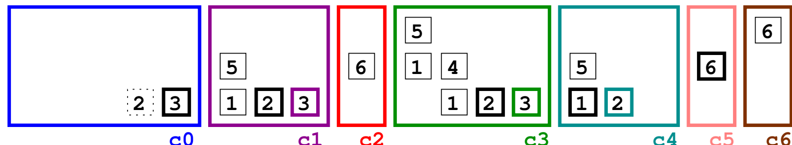
Retrait : c0-bloc 1, aucun export





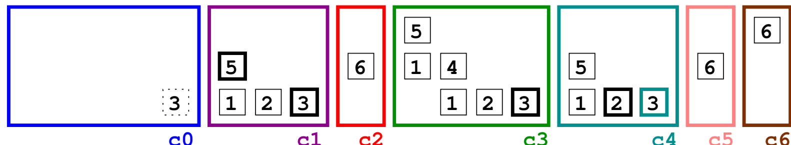
Retrait : c0-bloc 2, aucun export

```
rDS:  cmpq $2, %rsi                     1
      ja .L1

      movq (%rdi), %rax                 2
      jb .L2

      addq 8(%rdi), %rax                3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                         4
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                   5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                     6
      endfork
```



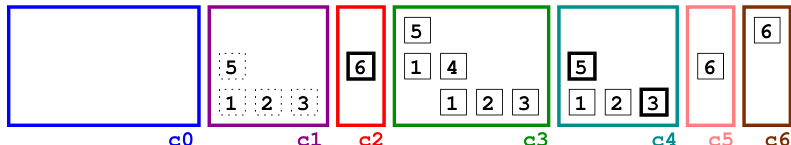
Retrait : c0-bloc 3, aucun export

```
rDS:   cmpq $2, %rsi                  1
       ja .L1

       movq (%rdi), %rax              2
       jb .L2

       addq 8(%rdi), %rax             3
.L2:   endfork

.L1:   subq $8, %rsp
       movq %rsi, %rbp                4
       shrq %rsi
       fork rDS

       movq %rax, 0(%rsp)
       leaq (%rdi, %rsi, 8), %rdi
       subq %rsi, %rbp                5
       movq %rbp, %rsi
       fork rDS

       addq 0(%rsp), %rax
       addq $8, %rsp                  6
       endfork
```



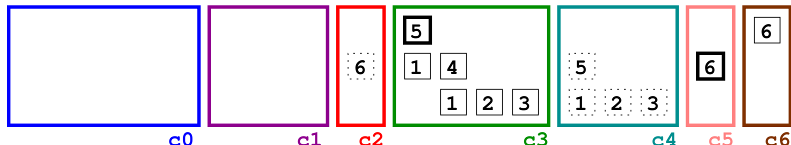
Retrait : c1-blocs 5,1,2,3, aucun export

```
rDS:  cmpq $2, %rsi                 1
      ja .L1
      movq (%rdi), %rax             2
      jb .L2
      addq 8(%rdi), %rax            3
.L2:  endfork
.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                     4
      fork rDS
      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp               5
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax
      addq $8, %rsp                 6
      endfork
```



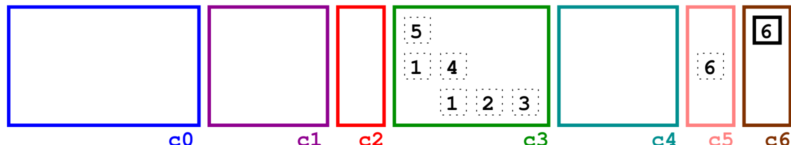
Retrait : c2-bloc 6 || c4-blocs 5,1,2,3, aucun export

```
rDS:  cmpq $2, %rsi                      1
      ja .L1
      movq (%rdi), %rax                  2
      jb .L2
      addq 8(%rdi), %rax                 3
.L2:  endfork
.L1:  subq $8, %rsp                      4
      movq %rsi, %rbp
      shrq %rsi
      fork rDS
      movq %rax, 0(%rsp)                 5
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp
      movq %rbp, %rsi
      fork rDS
      addq 0(%rsp), %rax                 6
      addq $8, %rsp
      endfork
```



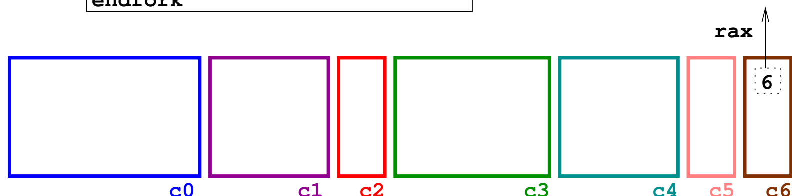
Retrait : c3-blocs 5,1,4,1,2,3 || c5-bloc 6, aucun export

```
rDS:  cmpq $2, %rsi                    1
      ja .L1

      movq (%rdi), %rax                2
      jb .L2

      addq 8(%rdi), %rax               3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp
      shrq %rsi                        4
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                  5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                    6
      endfork
```



Retrait : c6-bloc 6, export *rax*

```
rDS:  cmpq $2, %rsi                         1
      ja .L1

      movq (%rdi), %rax                     2
      jb .L2

      addq 8(%rdi), %rax                    3
.L2:  endfork

.L1:  subq $8, %rsp
      movq %rsi, %rbp                       4
      shrq %rsi
      fork rDS

      movq %rax, 0(%rsp)
      leaq (%rdi, %rsi, 8), %rdi
      subq %rsi, %rbp                       5
      movq %rbp, %rsi
      fork rDS

      addq 0(%rsp), %rax
      addq $8, %rsp                         6
      endfork
```



Conclusion.

Conclusion : ce qu'il faut retenir.

- Ordre total des exécutions des instructions de la trace (lecture et exécution séquentielles de C) :
  séquentiel + déterministe.

Conclusion : ce qu'il faut retenir.

- Ordre total des exécutions des instructions de la trace (lecture et exécution séquentielles de C) :
  séquentiel + déterministe.

- Ordre partiel + ordonnancement de l'OS (*pthread* ou *MPI* : lecture et exécution parallèles de *threads* ou *tasks*, synchronisations ou communications de l'OS) :
  parallèle (sync/comm OS) + non déterministe.

Conclusion : ce qu'il faut retenir.

- Ordre total des exécutions des instructions de la trace (lecture et exécution séquentielles de C) :
  séquentiel + déterministe.

- Ordre partiel + ordonnancement de l'OS (*pthread* ou *MPI* : lecture et exécution parallèles de *threads* ou *tasks*, synchronisations ou communications de l'OS) :
  parallèle (sync/comm OS) + non déterministe.

- Ordre total de la trace lue + ordre total de la trace renommée + ordre partiel *dataflow* des exécutions (lecture, renommage et exécution parallèles de C) :
  parallèle (sync/comm *dataflow*) + déterministe.

Conclusion : ce qu'il faut retenir.

- Ordre total des exécutions des instructions de la trace (lecture et exécution séquentielles de C) :
  séquentiel + déterministe.
- Ordre partiel + ordonnancement de l'OS (*pthread* ou *MPI* : lecture et exécution parallèles de *threads* ou *tasks*, synchronisations ou communications de l'OS) :
  parallèle (sync/comm OS) + non déterministe.
- Ordre total de la trace lue + ordre total de la trace renommée + ordre partiel *dataflow* des exécutions (lecture, renommage et exécution parallèles de C) :
  parallèle (sync/comm *dataflow*) + déterministe.
- http ://perso.numericable.fr/bernard.goossens/i_need_a_fork.html

"hello world" avec MPI : surcoût de l'OS.

```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init( &argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize();
}
```

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.