Cours 4 : Programmation collaborative

March 12, 2024

Séance 11 (S+4) ; Cours N+4 : Programmation collaborative

1) Introduction : Programmation collaborative (par patron/schéma élémentaire)
2) 1 prod - 1 conso
3) 3 tiers (1 prod - 1 consoProd - 1 conso et retour ?)
4) N prod - 1 conso
5) 1 prod - N conso
6) Autres
    - N prod - M conso ?
    - map-reduce en parallèle
    - mise en place des réseaux de communication (anneau, 4-complet, etc.)

1 Introduction

Quelques exemples de répartition d'un calcul sur un processeur multicoeur en utilisant plusieurs processus fonctionnant en parallèle selon des schémas élémentaires d'organisation.

Les points importants abordés :

- parallélisme réduit, à taille humaine (quelques processus seulement pour processeur multi-coeur début 21e siècle, i.e. le nombre de coeur se compte en unité, au maximum en dizaine)
- schéma simple (sans boucle de communication, type dataDriven, dataFlow)

1.1 Schémas usuels (liste rapide non exhaustive)

Rapidement, une liste des idées pour programmer en parallèle :

- modèle map - reduce (et proches : replicate - reduce, split - merge)
- modèle en tiers : 3-tiers, N-tiers
- modèle dataflow programming (et proches : Data-driven programming, Flow-based programming, Worflow programming, Stream Processing, Pipes, Pipeline)

plus loin (au delà de ce document)

- programmation vectorielle
- réseaux d'automates
- système multi-agents

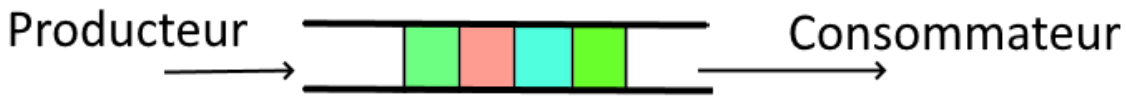
Pour la suite, une démarche "design pattern" (patron de conception / schémas élémentaire de programmation) sera adoptée. Ce qui signifie l'introduction d'un vocabulaire (ex. prod-conso), des dessins pour visualiser et des schémas de programmations (un peu) abstraits. Pour les mises en oeuvres pratiques, il faudra adapter.

En commentaire, sera donnée une indication des gains en temps de calcul espérés, avec une explicitation des limites du gain ; dans tous les cas, attention :

- la loi de Amdahl : https://fr.wikipedia.org/wiki/Loi_d%27Amdahl (peu de processeur, peu de gain sur temps global)
- la loi de Gustafson https://fr.wikipedia.org/wiki/Loi_de_Gustafson (plus de données, plus de gain sur temps moyen par donnée)

2 Schéma de base : 1 Producteur - 1 Consommateur

Le plus simple, le calcul est divisé en 2 , d'un côté un producteur de données intermédiaires envoyées de l'autre côté à un consommateur.



Gain visé :

- 2
- sous réserve que :
  - le temps de calcul du producteur soit proche du temps de calcul du consommateur
  - le débit de production des données intermédiaires soient proche du débit de consommation de ces données intermédiaires
  - la latence d'obtention des premières données intermédiaire soir négligeable (ou raisonnable)
  - le surcoût de la communication soit négligeable (ou raisonnable)

Un exemple :

- un processeur crée des données, cela lui prend un certains temps (lg)
- un consommateur utilise ces données, cela lui prend aussi du temps (lg)

[75]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

processusProducteur(PConso,0) -> PConso ! fin;
processusProducteur(PConso,P) ->
  lg(P),
  PConso ! P,
  processusProducteur(PConso,P-1).

processusConsommateur() ->
  receive C when is_integer(C) -> lg(C), processusConsommateur();
  fin -> io:format("Consommateur> fin. ~n",[]) end.

lg(0) -> done;
lg(N) -> lg(N div 2).
```

```
main([]) ->
  compile:file(prog),
  {ok, X} = io:read("Donne un nombre : "),
  io:format("~n *** 1 producteur - 1 consommateur *** ~n~n",[]),
  PrcsConso=spawn(prog,processusConsommateur,[]),
  _PrcsProcessusProd=spawn(prog,processusProducteur,[PrcsConso,X]),
  timer:sleep(5000).
```

```
Overwriting prog.erl
```

[77]:
```
!echo "1000." | escript prog.erl
```

```
Donne un nombre :
 *** 1 producteur - 1 consommateur ***

Consommateur> fin.
```

2.1 Composition simple (motif 3 tiers) : 1 Producteur - 1 ConsommateurProducteur - 1 Consommateur

En adoptant une décomposition en 3 tiers, le calcul est divisé en 3 , d'un côté un producteur d'une première série de données intermédiaires envoyées au centre à un consommateur-producteur qui transforme cette première série de données intermédiaires pour donner une seconde série de données intermédiaires envoyées à un consommateur final en bout de chaine.

Gain visé :

- 3
- sous réserve que :
  - le temps de calcul du producteur soit proche du temps de calcul du consommateur - producteur et proche du temps de calcul du consommateur final
  - le débit de production des données intermédiaires soient proche du débit de consommation de ces données intermédiaires
  - la latence d'obtention des données intermédiaire soir négligeable (ou raisonnable)
  - le surcoût de la communication soit négligeable (ou raisonnable)

Un exemple :

[139]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

processusProducteur(PConsoProd,0) -> PConsoProd ! fin;
processusProducteur(PConsoProd,P) ->
  lg(P),
  PConsoProd ! P,
  processusProducteur(PConsoProd,P-1).
```

```
processusConsommateurProducteur(PConso) ->
  receive C when is_integer(C) -> lg(C), PConso ! C,␣
 ↪processusConsommateurProducteur(PConso);
  fin -> PConso ! fin end.

processusConsommateur() ->
  receive C when is_integer(C) -> lg(C), processusConsommateur();
  fin -> io:format("Consommateur> fin. ~n",[]) end.

lg(0) -> done;
lg(N) -> lg(N div 2).


main([]) ->
  compile:file(prog),
  {ok, X} = io:read("Donne un nombre : "),
  io:format("~n *** 1 producteur - 1 consommateurProducteur - 1 consommateur␣
 ↪*** ~n~n",[]),
  PrcsConso=spawn(prog,processusConsommateur,[]),
  PrcsConsoProd=spawn(prog,processusConsommateurProducteur,[PrcsConso]),
  _PrcsProcessusProd=spawn(prog,processusProducteur,[PrcsConsoProd,X]),
  timer:sleep(5000).
```

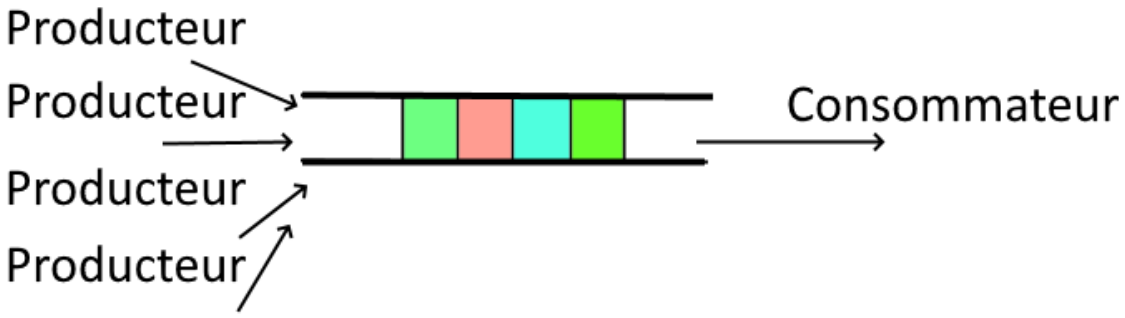
```
Overwriting prog.erl
```

[11]:
```
!echo "1000000." | escript prog.erl
```

```
Donne un nombre de producteurs : Donne un nombre de donnees : escript: exception
error: no match of right hand side value eof
  in function  erl_eval:expr/5 (erl_eval.erl, line 450)
  in call from escript:eval_exprs/5 (escript.erl, line 869)
  in call from erl_eval:local_func/6 (erl_eval.erl, line 572)
  in call from escript:interpret/4 (escript.erl, line 780)
  in call from escript:start/1 (escript.erl, line 277)
  in call from init:start_em/1
  in call from init:do_boot/3
```

2.2 N Producteurs - 1 Consommateur

Le calcul est divisé en N+1, avec N producteurs de séries de données intermédiaires envoyées à un consommateur.



Gain visé :

- N+1
- sous réserve que :
  - le temps de calcul des producteurs soit proche du temps de calcul du consommateur
  - le débit de production des données intermédiaires soient proche du débit de consommation de ces données intermédiaires
  - la latence d'obtention des données intermédiaire soir négligeable (ou raisonnable)
  - le surcoût de la communication soit négligeable (ou raisonnable)

Un exemple :

[90]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

processusNProducteurs(_PConso,0,_P) -> done;
processusNProducteurs(PConso,N,D) ->
  spawn(prog,processusProducteur,[PConso,D]),
  processusNProducteurs(PConso,N-1,D).

processusProducteur(_PConso,0) -> io:format("Producteur> fin. ~n",[]);
processusProducteur(PConso,D) ->
  lg(D),
  PConso ! D,
  processusProducteur(PConso,D-1).

processusConsommateur(N,D) when D < N -> io:format("Consommateur> fin. ~n",[]);
processusConsommateur(N,D) ->
  receive C -> lg(C), processusConsommateur(N,D-1) end.

lg(0) -> done;
lg(Z) -> lg(Z div 2).


main([]) ->
  compile:file(prog),
  {ok, N} = io:read("Donne un nombre de producteurs : "),
```

```
  {ok, D} = io:read("Donne un nombre de donnees : "),
  io:format("~n *** ~p producteurs - 1 consommateur (~p donnees)***␣
 ↪~n~n",[N,D]),
  PrcsConso=spawn(prog,processusConsommateur,[N,D]),
  processusNProducteurs(PrcsConso,N,D div N),
  timer:sleep(9000).
```

```
Overwriting prog.erl
```

[95]:
```
!echo -e "6.\n1000000." | escript prog.erl
```

```
Donne un nombre de producteurs : Donne un nombre de donnees :
 *** 6 producteurs - 1 consommateur (1000000 donnees)***

Producteur> fin.
Producteur> fin.
Producteur> fin.
Producteur> fin.
Producteur> fin.
Producteur> fin.
Consommateur> fin.
```

Rem. pour N prod => 1 Conso, la fin est (presque) facile à observer (il n'y a qu'un seul consommateur, donc une seule fin globale), on peut travailler à une meilleure forme de programme et de mesure du temps avec time :

[12]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

processusNProducteurs(_PConso,0,_P) -> done;
processusNProducteurs(PConso,N,D) ->
  spawn(prog,processusProducteur,[PConso,D]),
  processusNProducteurs(PConso,N-1,D).

processusProducteur(_PConso,0) -> io:format("Producteur> fin. ~n",[]);
processusProducteur(PConso,D) ->
  lin(D),  %%version avec bcp de calcul pour les producteurs
  PConso ! D,
  processusProducteur(PConso,D-1).

processusConsommateur(N,D,M) when D < N -> io:format("Consommateur> fin.␣
 ↪~n",[]), M ! fin;
processusConsommateur(N,D,M) ->
  receive C -> lg(C), processusConsommateur(N,D-1,M) end. %%version avec peu de␣
 ↪calcul pour le consommateur

lg(0) -> done;
```

```
lg(Z) -> lg(Z div 2).  %%version avec peu de calcul bcp de communication

lin(0) -> done;
lin(Z) -> lin(Z-1).    %%version avec bcp de calcul, peu de communication


main([]) ->
  compile:file(prog),
  {ok, N} = io:read("Donne un nombre de producteurs : "),
  {ok, D} = io:read("Donne un nombre de donnees : "),
  io:format("~n *** ~p producteurs - 1 consommateur (~p donnees)***␣
 ↪~n~n",[N,D]),
  PrcsConso=spawn(prog,processusConsommateur,[N,D,self()]),
  processusNProducteurs(PrcsConso,N,D div N),
  receive _X -> done end.
```

```
Overwriting prog.erl
```

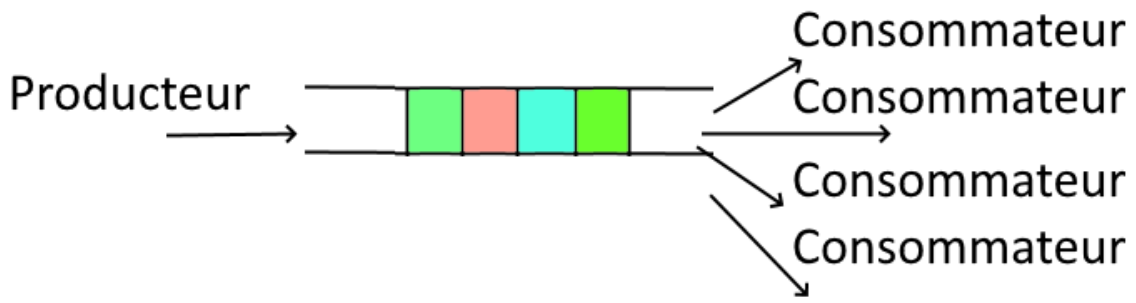
[29]:
```
!time echo -e "10.\n4000000." | escript prog.erl
```

```
Donne un nombre de producteurs : Donne un nombre de donnees :
 *** 10 producteurs - 1 consommateur (4000000 donnees)***

^C
```

2.3 1 Producteur - N Consommateurs

Le calcul est divisé en N+1, avec 1 producteurs d'une série de données intermédiaires envoyées à N consommateurs.



Gain visé :

- N+1
- sous réserve que :
  - le temps de calcul du producteurs soit proche du temps de calcul des consommateurs
  - le débit de production des données intermédiaires soient proche du débit de consommation de ces données intermédiaires
  - la latence d'obtention des données intermédiaire soir négligeable (ou raisonnable)
  - le surcoût de la communication soit négligeable (ou raisonnable)

Un exemple :

[111]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

processusProducteur(_LConsos,_LC,0) -> io:format("Producteur> fin. ~n",[]);
processusProducteur([],_LC,D) ->
 processusProducteur(_LC,_LC,D);
processusProducteur([PConso|LConsos],_LC,D) ->
  lg(D),
  PConso ! D,
  processusProducteur(LConsos,_LC,D-1).

procConsommateurs(0,_N) -> [];
procConsommateurs(N,NN) ->
  [spawn(prog,processusConsommateur,[NN])|procConsommateurs(N-1,NN)].

processusConsommateur(N) ->
  receive C when C =< N -> io:format("Consommateur> fin. ~n",[]);
          C -> lg(C), processusConsommateur(N) end.

lg(0) -> done;
lg(Z) -> lg(Z div 2).

main([]) ->
  compile:file(prog),
  {ok, N} = io:read("Donne un nombre de consommateur : "),
  {ok, D} = io:read("Donne un nombre de donnees : "),
  io:format("~n *** 1 producteurs - ~p consommateur (~p donnees)***␣
 ↪~n~n",[N,D]),
  ListeConsommateurs=procConsommateurs(N,N),
 ␣
 ↪_PrcsProcessusProd=spawn(prog,processusProducteur,[ListeConsommateurs,ListeConsommateurs,D]
  timer:sleep(9000).
```

```
Overwriting prog.erl
```

[113]:
```
!echo -e "6.\n1000000." | escript prog.erl
```

```
Donne un nombre de consommateur : Donne un nombre de donnees :
 *** 1 producteurs - 6 consommateur (1000000 donnees)***

Producteur> fin.
Consommateur> fin.
Consommateur> fin.
Consommateur> fin.
Consommateur> fin.
```

```
Consommateur> fin.
Consommateur> fin.
```

3 Autres

D'autres ?

- N prod - M conso ? (est-ce utile ?)
- map-reduce en parallèle **AFAIRE**
- mise en place des réseaux de communication
  - anneau **AFAIRE**
  - 4-complet **AFAIRE**