Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Parallélisme & Applications

Corrections TD n°5

Programmation MPI

Programmation linéaire

1 – Une société de livraison de fruits et légumes veut optimiser le traitement de ses produits lors de leur acheminement :

- chaque produit est stocké dans un carton identifié par un numéro, un prix au kilo et un poids ;
- un camion ne peut transporter au plus que 1000Kg ;

Les données du problème sont stockées dans un tableau à deux dimensions, dont l'indice correspond au numéro de carton :

Produit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	N
Poids	5	2	4	3	4	9	2	4	6	6	2	8	9	3	...	9
Prix	31	29	27	21	20	11	11	23	21	17	39	17	15	43	...	35

L'optimisation consiste à explorer **toutes les possibilités** de chargement du camion en maximisant le produit de sa vente :

i. on construira une proposition de chargement :
On utilisera une relation d'appartenance pour exprimer une proposition de chargement de tous les cartons :

Produit	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	...	N
Chargé	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*1*	*0*	*0*	*1*	...	*0*

ii. vérifier que le chargement est possible, c-à-d que la somme des poids est inférieure à 1000Kg ;
iii. si le chargement est possible, évaluer le prix global de la proposition ;
iv. recommencer pour toutes les propositions possibles et sélectionner celle qui a un prix global le plus élevé.

Questions :

a. Combien de **propositions** doivent être construites sachant qu'il y a N cartons ?
Pour chaque produit, il y a deux choix possibles : le charger ou non. Il y a donc 2^N possibilités de chargement.
Quelle est la **complexité** de l'algorithme de génération ?
La complexité est en $o(2^n)$.

b. Décrivez les **structures de données** que vous utiliserez pour trouver le meilleur chargement.
On va utiliser une représentation compacte de nos combinaisons, en utilisant des entiers non signés où chaque bit indique si le produit associé par son rang est chargé ou non.
On utilisera une représentation Big Endian : $2^{(N-1)}$ associé au produit 1, $2^{(N-2)}$ associé au produit 2, etc.
Pour rendre la taille de notre représentation indépendante de la plus grande taille d'entier non signé, on utilisera un tableau d'entiers signés.

c. Donnez une méthode de **génération parallèle** de toutes les propositions de chargement.
*Tout d'abord, il nous faut un mécanisme capable de construire toutes les combinaisons possibles des bits $\Longrightarrow$ on va utiliser un **compteur étendu** sur un tableau d'entiers.*
Pour répartir de manière équitable le travail entre les différents nœud de la machine parallèle, on va utiliser un préfixe binaire pour notre chargement :
- *imaginons que l'on ait 4 processeurs, soient un préfixe sur 2 bits :*

- ⋆ *le nœud 0 reçoit le préfixe `00` ⟹ c-à-d que toutes les chargements qu'il va explorer ne contiendront jamais les deux premiers produits ;*
- ⋆ *le nœud 1 reçoit le préfixe `01` ⟹ c-à-d que toutes les chargements qu'il va explorer ne contiendront jamais le premier produit, mais toujours le second ;*
- ⋆ *le nœud 2 reçoit le préfixe `10` ⟹ c-à-d que toutes les chargements qu'il va explorer ne contiendront jamais le second produit, mais toujours le premier ;*
- ⋆ *le nœud 3 reçoit le préfixe `11` ⟹ c-à-d que toutes les chargements qu'il va explorer contiendront toujours les deux premiers produits ;*
  On remarque que le préfixe binaire correspond au numéro du nœud, et que dans MPI les nœuds sont automatiquement numérotés.
  *La répartition du travail est **parfaite** : chaque nœud reçoit un nombre identique de combinaisons à explorer.*
  Par contre il faut que le nombre de nœuds soit une puissance de 2.

La complexité de cette méthode est-elle **optimale** ?
Le coût du est optimal si $c(n) = t(n) * p(n)$ avec $c(n)$ similaire en complexité à la version séquentielle.
Ici, $t(n) = o(\frac{2^N}{N})$ et $c(n) = o(n) \Longrightarrow$ coût optimal.

Est-ce du parallélisme de contrôle ou de données ?
C'est du parallélisme de données : on explore en parallèle l'espace des combinaisons correspond à tous les chargements possibles.

d. Donnez une **méthode parallèle** de :
   - ◇ vérification de la faisabilité du chargement
   - ◇ calcul du prix global de vente.

```
#ifndef REPRESENTATION_BINAIRE_H
#define REPRESENTATION_BINAIRE_H

#include <inttypes.h>

#define TYPE uint8_t
#define TYPE_MPI MPI_UINT8_T
#define NBBITSTYPE 8
#define TAILLE 20
#define POIDS_MAX 90

#define NBWORDS ((TAILLE+NBBITSTYPE-1)/NBBITSTYPE)
#define RANGWORDMAX (NBWORDS-1)
#define UN(c,rang) (c->tab[RANGWORDMAX-rang/NBBITSTYPE] = c->tab[RANGWORDMAX-
rang/NBBITSTYPE]|1<<(rang%NBBITSTYPE))
#define ZERO(c,rang) (c->tab[RANGWORDMAX-rang/NBBITSTYPE] = c->tab[RANGWORDMAX-
rang/NBBITSTYPE]&~((TYPE)1<<(rang%NBBITSTYPE)))
#define GET(c,rang) ((c->tab[RANGWORDMAX-rang/NBBITSTYPE]&(1<<rang%NBBITSTYPE))?1:0)

typedef struct {
  TYPE *tab;
  uint32_t taille;
} combinaison;

extern TYPE ValeurMaxPremier;
TYPE rb_calculer_valeur_max(uint32_t);
combinaison *rb_allouer(uint32_t);
combinaison *rb_liberer(combinaison *);
/* rb_copier(destination, source) */
void rb_copier(combinaison *, combinaison *);
void rb_tester();
int  rb_incrementer(combinaison *);
void rb_afficher(combinaison *);

#endif
```

On peut changer le type des éléments du tableau binaire et par exemple utiliser des `uint64_t` qui offrent 64bits.
Ici, pour des raisons de vérification du bon fonctionnement, c'est le type le plus petit qui a été choisi : `uint8_t`.
Il faut également faire correspondre le type choisi au type MPI à utiliser pour les communications.
On peut également choisir les paramètres : nombre de produits à gérer et poids à ne pas dépasser.
Des macros sont utilisées pour accéder à chaque bit et les modifier.

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <assert.h>

#include "representation_binaire.h"

TYPE ValeurMaxPremier = 0;

combinaison *rb_allouer(uint32_t taille)
{
  combinaison *c = malloc(sizeof(combinaison));
  c->tab = calloc(NBWORDS,sizeof(TYPE));
  c->taille = taille;

  return c;
}

combinaison *rb_liberer(combinaison *c)
{
  free(c->tab);
  free(c);

  return NULL;
}

void rb_copier(combinaison *dst, combinaison *src)
{
  assert(src->taille == dst->taille);
  for(int i = 0; i < src->taille; i++)
    dst->tab[i] = src->tab[i];
}

TYPE rb_calculer_valeur_max(uint32_t taille)
{
  TYPE valeur_max = 0;
  uint32_t rang_dernier_bit = (taille % NBBITSTYPE) ? taille% NBBITSTYPE : 8;

  for(int i=0;i<rang_dernier_bit; i++)
    valeur_max += 1<<i;

  return valeur_max;
}

void rb_afficher(combinaison *c)
{
  for(int i = 0; i <= RANGWORDMAX; i++)
    printf("%" PRIu64 ", ", (uint64_t)c->tab[i]);
  printf("\n");
}

int rb_incrementer(combinaison *c)
{
  int i;
  TYPE tester_max = -1;

  for(i = RANGWORDMAX; i >= 0; i--)
  {
    c->tab[i]++;
    if (c->tab[i] == 0)
      continue;
    else
      break;
  }
  if (c->tab[0] == ValeurMaxPremier)
  {
    int condition = 1;
    for(int i = RANGWORDMAX; i >= 1; i--)
    {
      condition = condition && c->tab[i] == tester_max;
      if (condition == 0)
        return 0;
    }
    return 1;
  }
  return 0;
}
```

Les combinaisons sont gérées sous forme de tableau de « bits *», avec une taille max à utiliser.*
Il y a également la fonction servant à incrémenter qui retourne 1 si elle a atteint la valeur max.

```c
#ifndef GESTION_CHARGEMENT_H
#define GESTION_CHARGEMENT_H
#include "representation_binaire.h"

typedef struct {
    int prix;
    int poids;
} produit;

extern produit liste_produits[];

void gc_afficher_produits();
/* gc_evaluer_chargement(&combinaison, &prix, &poids) */
void gc_evaluer_chargement(combinaison *, int *, int *);
/* gc_evaluer_prefixe(taille_prefixe, valeur_prefixe, &prix, &poids) */
void gc_evaluer_prefixe(int, int, int *, int *);
/* gc_afficher_chargement(combinaison, taille_prefixe, valeur_prefixe) */
void gc_afficher_chargement(combinaison *, int, int );
#endif
```

```c
#include <stdio.h>
#include "gestion_chargement.h"

produit liste_produits[] = {                    // Tableau genere par l'IA Grok v2
  {15, 2}, {20, 3}, {30, 5}, {10, 1}, {40, 6},
  {50, 7}, {25, 4}, {60, 8}, {70, 9}, {35, 5},
  {80, 10}, {90, 11}, {45, 6}, {100, 12}, {110, 13},
  {55, 7}, {120, 14}, {130, 15}, {65, 8}, {140, 16},
  {150, 17}, {75, 9}, {160, 18}, {170, 19}, {85, 10},
  {180, 20}, {190, 21}, {95, 11}, {200, 22}, {210, 23},
  {105, 12}, {220, 24}, {230, 25}, {115, 13}
};

void gc_afficher_produits() {
  int taille = sizeof(liste_produits) / sizeof(produit);

  for (int i = 0; i < taille; i++) {
    printf("Produit %d: Prix = %d, Poids = %d\n", i,
        liste_produits[i].prix, liste_produits[i].poids);
  }
}

void gc_evaluer_chargement(combinaison *c, int *prix_chargement, int *poids_chargement)
{
  for(int i = c->taille - 1; i >= 0; i--)
    if GET(c,i) {
      *prix_chargement += liste_produits[TAILLE-i-1].prix;
      *poids_chargement += liste_produits[TAILLE-i-1].poids;
  }
}

void gc_evaluer_prefixe(int taille_prefixe, int valeur_prefixe, int *prix_prefixe, int
*poids_prefixe)
{
  for(int i = TAILLE - taille_prefixe; i < TAILLE; i++)
  {
    if (valeur_prefixe & 1)
    {
      /* printf("Ajout du produit %d, poids : %d, prix : %d\n", i, lp[TAILLE-1-i].poids,
lp[TAILLE-1-i].prix); */
      *poids_prefixe += liste_produits[TAILLE - 1  - i].poids;
      *prix_prefixe += liste_produits[TAILLE - 1 - i].prix;
    }
    valeur_prefixe = valeur_prefixe >> 1;
  }
}

void gc_afficher_chargement(combinaison *c, int taille_prefixe, int valeur_prefixe)
{
  for(int i = TAILLE - taille_prefixe; i < TAILLE; i++)
  {
    if (valeur_prefixe & 1)
      UN(c,i);
    valeur_prefixe = valeur_prefixe >> 1;
  }

  for(int i = c->taille + taille_prefixe - 1; i >= 0; i--)
    if GET(c,i) {
      printf("[%d, prix: %d, poids : %d]\n", i, liste_produits[TAILLE-i-1].prix,
liste_produits[TAILLE-i-1].poids);
  }
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include "gestion_chargement.h"
#include "representation_binaire.h"

#define NB_NOEUDS 4 // soient 2 bits de prefix

int main()
{
  int rank, nprocs;
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (nprocs != NB_NOEUDS)
  {     printf("Vous devez lancer %d noeuds ou modifier le programme\n", NB_NOEUDS);
        MPI_Finalize();
        return 1;       }

  if (rank == 0) gc_afficher_produits();
  MPI_Barrier(MPI_COMM_WORLD);
  uint32_t taille_prefixe = log2(NB_NOEUDS);
  int poids_prefixe = 0, prix_prefixe = 0;
  /* On calcule le poids et le prix du prefixe */
  gc_evaluer_prefixe(taille_prefixe, rank, &prix_prefixe, &poids_prefixe);
  printf("Proc %d : prix prefixe : %d poids prefixe : %d\n", rank, prix_prefixe, poids_pre
fixe);

  /* On calcule la valeur max du premier element de la combinaison */
  rb_calculer_valeur_max(TAILLE);
  combinaison *combinaison_travail;
  combinaison *combinaison_max;
  combinaison_travail = rb_allouer(TAILLE-taille_prefixe);
  combinaison_max = rb_allouer(TAILLE-taille_prefixe);
  int prix_max = 0, poids_max = 0;
  int prix_total = 0, poids_total = 0;
  int fini = 0;

  do {  prix_total = prix_prefixe;
        poids_total = poids_prefixe;
        fini = rb_incrementer(combinaison_travail);
        gc_evaluer_chargement(combinaison_travail, &prix_total, &poids_total);
        if ((poids_total < POIDS_MAX) && (prix_total > prix_max))
        {
          prix_max = prix_total;
          poids_max = poids_total;
          rb_copier(combinaison_max, combinaison_travail);
        }
  } while(!fini);

  printf("Proc: %d, evaluation finale : prix : %d, poids : %d\n", rank, prix_max, poids_max);

  if (rank == 0)
  {
        MPI_Status status;
        combinaison *combinaison_noeud = rb_allouer(TAILLE - taille_prefixe);

        /* On recupere les resultats des autres noeuds */
        for (int i = 1; i < NB_NOEUDS; i++)
        { int prix_noeud = 0;
          int poids_noeud = 0;
          MPI_Recv (combinaison_noeud->tab,    /* address of receive buffer */
                  TAILLE - taille_prefixe,      /* number of items to receive */
                  TYPE_MPI,  /* type of data */
                  MPI_ANY_SOURCE, /* can receive from any other */
                  1,      /* tag */
                  MPI_COMM_WORLD, /* communicator */
                  &status);   /* status  */
          gc_evaluer_prefixe(taille_prefixe, status.MPI_SOURCE, &prix_noeud, &poids_noeud);
          gc_evaluer_chargement(combinaison_noeud, &prix_noeud, &poids_noeud);
          gc_afficher_chargement(combinaison_noeud, taille_prefixe, status.MPI_SOURCE);
          printf("Proc: %d, evaluation finale : prix : %d, poids : %d\n", status.MPI_SOURCE,
prix_noeud, poids_noeud);
        }
  }
  else
  {     MPI_Send (combinaison_max->tab, TAILLE - taille_prefixe,
                TYPE_MPI, /* sending ints */
                0,    /* to 0  */
                1,      /* tag */
                MPI_COMM_WORLD); /* communicator */
  }
  MPI_Finalize();
}
```

Il ne reste plus qu'à déterminer le max à chaque réception d'une combinaison de la part d'un nœud.

Est-il possible de proposer une méthode efficace améliorant l'équilibrage de charge de travail des différents cœurs ?
Comme on l'a vu, si on dispose d'une puissance de 2 comme nombre de nœuds on a un équilibrage de charge parfait.
Est-ce que les « *tasks* » sont intéressantes ?
Cela sous-entend que l'on utilise openMP, par exemple pour faire la somme des poids et prix d'une combinaison.
Dans ce cas, un « omp parallel for *» est le plus efficace et ne nécessite pas de «* tasks *» (le travail à réaliser est régulier et l'utilisation de «* tasks *» conduirait à une surcharge de travail inutile.*

Tester le projet

Le source de cette correction est disponible à `https://git.p-fb.net/PeFClic/parapp_td_5`

Attention : pour cause d'incompatibilité avec Ubuntu 24.04, l'exécution se fait avec l'outil `prterun`, disponible à `https://github.com/openpmix/prrte`