

Programmation MPI

■ ■ ■ **Modèle Master/Worker**

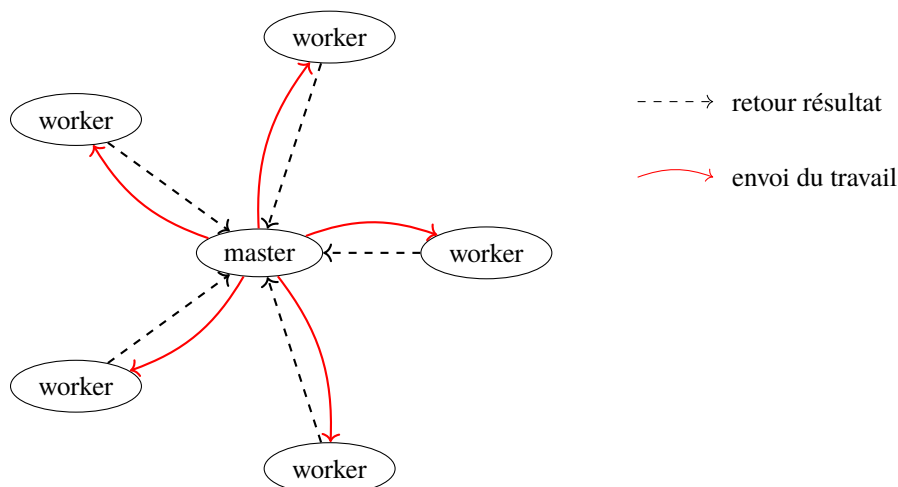
Un problème à paralléliser

Calculer la somme d'une liste de nombres entiers stockés dans un tableau.

Supposons qu'il y ait 100 entiers stockés dans le tableau « `elements` ».

Comment allons-nous paralléliser ce programme? C'est-à-dire, comment allons nous procéder, de telle sorte que ce problème soit résolu par plusieurs programmes s'exécutant simultanément, de façon parallèle.

Une solution : le **modèle Master/Workers**



Supposons quatre processeurs ou quatre processus qui vont travailler de façon simultanée à la résolution du problème.

La méthode la plus simple consiste alors à scinder le tableau en quatre parties, le traitement de chacune étant confié à un processus donné.

En conséquence, la parallélisation de ce problème s'effectue comme suit :

- quatre processus, que nous appellerons P_0, P_1, P_2, P_3 résoudront l'ensemble du problème ;
- la répartition des données est la suivante : P_0 calculera la somme des éléments 0 à 24 du tableau, P_1 celle des éléments 25 à 49, P_2 de 50 à 74 et enfin P_3 de 75 à 99 ;
- quand ces processus se seront exécutés, il faudra un autre processus qui effectuera la somme des quatre résultats intermédiaires afin de calculer la solution du problème ;
- les éléments du tableau ne sont pas connus des processus $P_0 \dots P_3$, et par conséquent, un **autre processus** doit également leur passer les valeurs de ces mêmes éléments ;

Il est nécessaire, en plus des processus P_0 à P_3 , de disposer d'un **cinquième processus** chargé de **distribuer** les données, **collecter** les résultats et **synchroniser** les exécutions.

Un tel processus est appelé processus *maître*, et les processus P_0 à P_3 sont appelés processus *worker*.

■ ■ ■ Exemple d'application en modèle Maître/Workers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4
5 void mastercode (int n,int m);
6 void workercode (void);
7
8 int main (int argc, char *argv[])
9 { int rank, nprocs;
10  MPI_Init (&argc, &argv);
11  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
12  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
13
14  if (argc != 3)
15  {
16      if (rank == 0)
17      {
18          printf("Please call this program with to arguments: N and M\n");
19          printf("The program will than calculate sum(1..N)\n");
20          printf("Each worker will calculate the sum of M numbers\n");
21      }
22      MPI_Finalize ();
23      return 1;
24  }
25
26  if (rank == 0)
27  { int n,m;
28      n = atoi (argv[1]);
29      m = atoi (argv[2]);
30      mastercode (n,m);
31  }
32  else workercode ();
33  MPI_Finalize ();
34  return 0;
35 }
```

```

1 void mastercode (int n,int m)
2 { int i;
3   long long answer;
4   int who, nprocs;
5   int task[2];
6   MPI_Status status;
7   long long sum = 0;
8   int answers_to_receive, received_answers;
9   int *counts;
10  int whomax,num;
11
12  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
13
14  answers_to_receive = (n - 1)/m + 1;
15  received_answers = 0;
16  num = 1;
17
18  /* send tasks to workers */
19  whomax = nprocs-1;
20  if (whomax > answers_to_receive) whomax = answers_to_receive;
21  for (who=1; who<=whomax; who++)
22  { task[0] = num;
23    task[1] = num + m - 1;
24    if (task[1] > n) task[1] = n;
25    MPI_Send (&task[0], 2, MPI_INT, /* sending two ints */
26              who, /* to the lucky one */
27              1, /* tag */
28              MPI_COMM_WORLD); /* communicator */
29    num += m;
30  }
31  while (received_answers < answers_to_receive)
32  { /* wait for an answer from a worker. */
33    MPI_Recv (&answer, /* address of receive buffer */
34             1, /* number of items to receive */
35             MPI_LONG_LONG, /* type of data */
36             MPI_ANY_SOURCE, /* can receive from any other */
37             1, /* tag */
38             MPI_COMM_WORLD, /* communicator */
39             &status); /* status */
40
41    who = status.MPI_SOURCE; /* find out who sent us the answer */
42    sum += answer; /* update the sum */
43    received_answers++; /* and the number of received answers */
44
45    /* put the worker on work, but only if not all tasks have been sent.
46     * we use the value of num to detect this */
47    if (num <= n)
48    { task[0] = num;
49      task[1] = num + m - 1;
50      if (task[1] > n)
51        task[1] = n;
52      MPI_Send (&task[0], 2, MPI_INT, /* sending two ints */
53                who, /* to the lucky one */
54                1, /* tag */
55                MPI_COMM_WORLD); /* communicator */
56      num += m;
57    }
58  }
59  /* Now master sends a message to the workers to signify that they should
60   * end the calculations. We use a special tag for that:
61   */
62  counts = (int*) malloc(sizeof(int)*(nprocs-1));
63  for (who = 1; who < nprocs; who++)
64  { MPI_Send (&task[0], 1, MPI_INT, /* sending one int
65    * it is permitted to send a shorter message than will be received.
66    * The other case:
67    * sending a longer message than the receiver expects is not allowed. */
68            who, /* to who */
69            2, /* tag */
70            MPI_COMM_WORLD); /* communicator */
71
72    /* the worker will send to master the number of calculations
73     * that have been performed. We put this number in the counts array.
74     */
75    MPI_Recv (&counts[who-1], /* address of receive buffer */
76             1, /* number of items to receive */
77             MPI_INT, /* type of data */
78             who, /* receive from process who */
79             7, /* tag */
80             MPI_COMM_WORLD, /* communicator */
81             &status); /* status */
82  }
83  printf ("The sum of the integers from 1..%d is %lld\n", n, sum);
84  printf ("The work was divided using %d summations per worker\n",m);
85  printf (" WORKER calculations\n\n");
86  for (i = 1; i < nprocs; i++)
87    printf ("%6d %8d\n", i, counts[i-1]);
88 }

```

```

1 void workercode ()
2 {
3     int i,rank;
4     long long answer;
5     int task[2];
6     MPI_Status status;
7     int count = 0;
8
9     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
10
11     /* worker first enters 'waiting for message' state
12      */
13
14     MPI_Recv (&task[0],          /* address of receive buffer */
15              2,                  /* number of items to receive */
16              MPI_INT,            /* type of data */
17              0,                  /* can receive from master only */
18              MPI_ANY_TAG,        /* can expect two values, so
19                                   * we use the wildcard MPI_ANY_TAG
20                                   * here */
21              MPI_COMM_WORLD,     /* communicator */
22              &status);           /* status */
23
24     /* if tag equals 2, then skip the calculations */
25
26     if (status.MPI_TAG !=2)
27     {
28         while (1)
29         {
30             answer = 0;
31
32             for (i = task[0]; i <= task[1]; i++)
33                 answer += i;
34             count++;
35             MPI_Send (&answer, 1, /* sending one int */
36                      MPI_LONG_LONG, 0, /* to master */
37                      1, /* tag */
38                      MPI_COMM_WORLD); /* communicator */
39
40             MPI_Recv (&task[0], /* address of receive buffer */
41                      2, /* number of items to receive */
42                      MPI_INT, /* type of data */
43                      0, /* can receive from master only */
44                      MPI_ANY_TAG, /* can expect two values, so
45                                   * we use the wildcard MPI_ANY_TAG
46                                   * here */
47                      MPI_COMM_WORLD, /* communicator */
48                      &status); /* status */
49
50             if (status.MPI_TAG == 2) /* leave this loop if tag equals 2 */
51                 break;
52
53         }
54     }
55
56     /* this is the point that is reached when a task is received with
57      * tag = 2 */
58
59     /* send the number of calculations to master and return */
60
61     MPI_Send (&count, 1, MPI_INT, /* sending one int */
62              0, /* to master */
63              7, /* tag */
64              MPI_COMM_WORLD); /* communicator */
65 }

```

- 1 – a. Commentez l'exemple : que fait-il ?
b. Quelle est la structure de communication ?

■ ■ ■ Utilisation de la diffusion & de la réduction

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main (int argc, char *argv[])
5 { int nprocs;          /* number of processes */
6   int rank;            /* the unique identification of this process */
7   int i, mysum, sum, imin, imax;
8   const int N = 60;
9
10  /* initialize the MPI environment: */
11  MPI_Init (&argc, &argv); /* note: use argc and argv yourself only after this call */
12  MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* rank will be different for all processes */
13  MPI_Comm_size (MPI_COMM_WORLD, &nprocs); /* nprocs will be the same for all processes */
14  printf ("Hello, this is process %d of a total of %d\n", rank, nprocs);
15
16  /* Here follows the first example of a real parallel program:
17   * We want to compute the sum of the first 60 integers.
18   * Each process will perform part of the work: for example
19   * with 4 processes: process 0 will sum up the numbers 1 to 15,
20   * process 1 the numbers 16 to 30, and so on.
21   * After this, the partial sums are summed up and the result
22   * is printed. For simplicity we assume that N is divisible by the number
23   * of processes, so which numbers are to be added in this process: */
24
25  imin = (N / nprocs) * rank + 1;
26  imax = imin + N / nprocs - 1;
27  mysum = 0; /* mysum will contain the sum of the numbers imin .. imax */
28  for (i = imin; i <= imax; i++)
29    mysum += i;
30
31  /* Now, each process has it's own partial sum: mysum
32   * We want that the process with rank equal to 0 will receive the
33   * sum of the partials sums. MPI_Reduce is used for this: */
34  MPI_Reduce (&mysum, /* the partial sum */
35             &sum, /* the total sum */
36             1, /* the number of elements in sum, in this case one int */
37             MPI_INT, /* the type of sum: int. Other types are for example:
38                      * MPI_DOUBLE
39                      * MPI_FLOAT */
40             MPI_SUM, /* we want to sum the partial sums. Other
41                      * operations are for example:
42                      * MPI_MAX find the maximum value
43                      * MPI_MIN find the minimum value
44                      * MPI_PROD find the product
45                      */
46             0, /* the rank of the process that is going to receive the sum. */
47             MPI_COMM_WORLD /* the communicator */
48  );
49  /* Now the sum of all mysum's is available on process 0 in variable sum */
50  if (rank == 0)
51    printf ("process zero reports: sum is %d\n", sum);
52
53  /* Let's try to communicate the value of sum to all processes
54   * using MPI_Bcast, a broadcasting subroutine: */
55  MPI_Bcast (&sum, /* the value to broadcast */
56            1, /* number of elements in sum */
57            MPI_INT, /* the datatype of sum */
58            0, /* the rank of the process that contains the value to broadcast */
59            MPI_COMM_WORLD /* the communicator */
60  );
61  /* Note that ALL processes execute the same call to MPI_Bcast.
62   * However, in this case the effect is that process 0 is sending
63   * and processes 1..nproc-1 are receiving. */
64  printf ("process %d reports: mysum is %d and sum is %d\n", rank, mysum, sum);
65
66  /* In practice, one would use MPI_Allreduce in stead of
67   * MPI_Reduce followed by MPI_Bcast. MPI_Allreduce is more
68   * efficient and leads to a shorter program: */
69  MPI_Allreduce (&mysum, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
70
71  /* Note that MPI_Allreduce is even more simple to use than
72   * MPI_Reduce: because the result is sent to every process,
73   * there is no need to specify to which process the answer
74   * must go */
75  printf ("process %d reports again: mysum is %d and sum is %d\n", rank, mysum, sum);
76
77  MPI_Finalize (); /* end of MPI. Do not forget this call: if a MPI
78                  * program ends without calling MPI_Finalize(),
79                  * strange things can happen ... */
80  return 0;
81 }
```

- 2 – a. Commentez l'exemple : que fait-il ?
b. Quelle est la structure de communication ?