البرمجة التفرعية باستخدام الـ MPI ضمن بيئة الـ VS2010

MPI Broadcast and Collective Communication

1 مفردات الجلسة:

- ✓ أنماط الاتصال ضمن الـ MPI والتوابع المستخدمة
- ✓ تدريب عملي

2 أنماط الاتصال ضمن الـ MPI والتوابع المستخدمة

1.2 أنماط الاتصال:

يوجد العديد من أنماط الاتصال ضمن تقنية الـ MPI منها الاتصال من نقطة إلى نقطة، وهو الاتصال بين عمليتين والتي تتم عن طريق الإجرائيتين send, receive. يوجد نمط أخر وهو الاتصال الجماعي (collective communication).

الاتصال الجماعي هو وسيلة اتصال تتضمن مشاركة جميع العمليات في جهاز التواصل. يمكن تصنيف هذا النمط إلى صنفين هما الروتين الجماعي القياسي (standard collective routine) و البث العام (broadcasting)

2.2 نقاط الاتصال والتزامن الجماعي

أحد الأشياء التي يجب تذكرها حول الاتصال الجماعي هو أنه يتضمن نقطة تزامن بين العمليات. وهذا يعني أن جميع العمليات يجب أن تصل إلى نقطة في التعليمات البرمجية الخاصة بها قبل أن تتمكن من البدء في التنفيذ مرة أخرى.

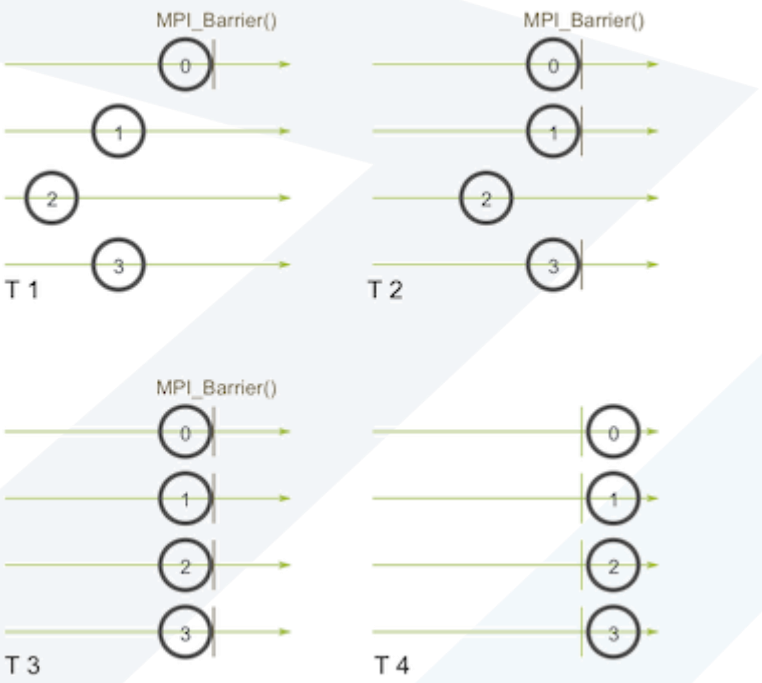
تحتوي الـ MPI على تابع خاص مخصص لمزامنة العمليات:

```
MPI_Barrier(MPI_Comm communicator)
```

يقوم هذا التابع بتشكيل حاجزًا، ولا يمكن لأي عملية في جهاز الاتصال تجاوز الحاجز حتى تستدعي جميعها التابع.

مثال توضيحي. بفرض أن المحور الأفقي يمثل تنفيذ البرنامج والدوائر تمثل عمليات مختلفة:

تستدعي العملية صفر أولاً MPI_Barrier في اللحظة الأولى (T 1). في حين أن العملية صفر معلقة عند الحاجز، فإن العملية الأولى والثالثة تصلان في النهاية (T 2). عندما تصل العملية الثانية أخيرًا إلى الحاجز (T 3)، تبدأ جميع العمليات في التنفيذ مرة أخرى (T 4).



أحد الاستخدامات الأساسية لـ MPI_Barrier هو مزامنة البرنامج بحيث يمكن توقيت تنفيذ أجزاء من الكود المتوازي بدقة.

إذا لم تتمكن من إكمال MPI_Barrier بنجاح، فلن تتمكن أيضًا من إكمال أي عملية اتصال جماعية بنجاح. إذا حاولت الاتصال بـ MPI_Barrier أو إجراءات جماعية أخرى دون التأكد من أن جميع العمليات في جهاز الاتصال ستستدعيه أيضًا، فسيتوقف تنفيذ البرنامج.

3.2 البث العام باستخدام MPI_Bcast

يعتبر البث العام إحدى تقنيات الاتصال الجماعي القياسية. تقوم إحدى العمليات بإرسال نفس البيانات إلى جميع العمليات في جهاز الاتصال. أحد الاستخدامات الرئيسية للبث هو إرسال مدخلات المستخدم إلى برنامج موازٍ، أو إرسال معلمات التكوين إلى جميع العمليات.

في الـ MPI، يمكن إجراء البث باستخدام MPI_Bcast. يبدو النموذج الأولي للوظيفة كما يلي:

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

على الرغم من أن العملية التي تنفذ في عقدة الجذر والعقدة المستقبلة تقومان بمهام مختلفة، إلا أنهما جميعًا يستدعيان نفس التابع MPI_Bcast. عندما تستدعي عقدة الجذر (مثلاً العملية صفر) MPI_Bcast، سيتم إرسال متغير البيانات إلى كافة العمليات الأخرى. عندما تستدعي كافة عمليات العقد المستقبلة MPI_Bcast، سيتم ملء متغير البيانات بالبيانات.

4.2 البث العام باستخدام MPI_Send وMPI_Recv

يمكن تمثيل التابع MPI_Bcast كما يلي:

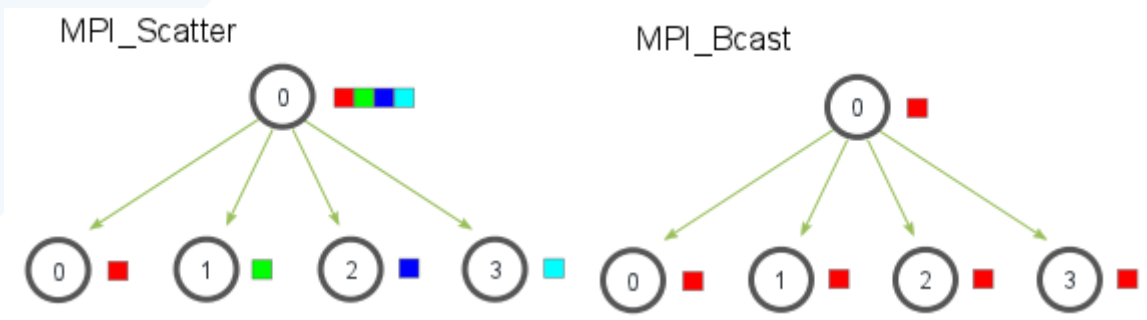
```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root
,                MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```

في الواقع هذا التابع غير فعال للغاية! يتمثل هذا التابع في استخدام رابط شبكة واحد فقط من العملية صفر لإرسال جميع البيانات. التنفيذ الأكثر ذكاءً هو خوارزمية اتصال قائمة على الشجرة يمكنها استخدام المزيد من روابط الشبكة المتاحة في وقت واحد.

5.2 الروتين الجماعي القياسي (standard collective routine)

تتم عن طريق استخدام التابعين MPI_Scatter and MPI_Gather.

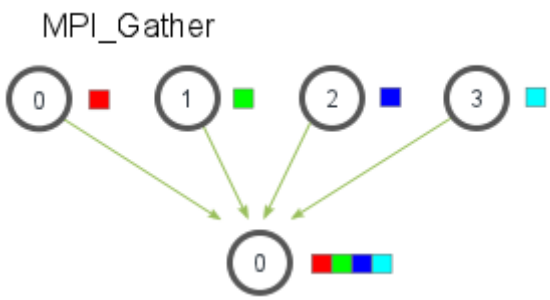
MPI_Scatter يشبه إلى حد كبير MPI_Bcast. يتضمن MPI_Scatter عملية جذرية معينة ترسل البيانات إلى جميع العمليات في جهاز الاتصال. الفرق الأساسي بين MPI_Bcast وMPI_Scatter صغير ولكنه مهم. يرسل MPI_Bcast نفس جزء البيانات إلى جميع العمليات بينما يرسل MPI_Scatter أجزاء من مصفوفة إلى عمليات مختلفة. يمكن توضيح ذلك بالشكل التالي:



تمتلك الصيغة التالية:

```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

MPI_Gather هو عكس MPI_Scatter. بدلاً من ارسال العناصر من عملية واحدة إلى العديد من العمليات، يستقبل MPI_Gather عناصر من العديد من العمليات ويجمعها في عملية واحدة. يعد هذا الروتين مفيدًا جدًا للعديد من الخوارزميات المتوازية، مثل الفرز المتوازي والبحث. وفيما يلي توضيح بسيط لهذه الخوارزمية.



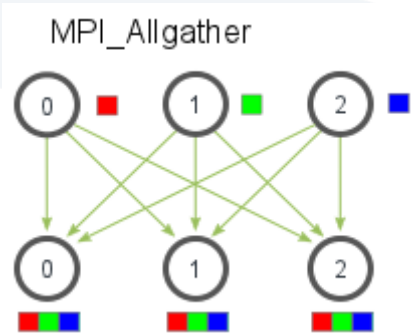
تمتلك الصيغة التالية:

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

MPI_Allgather: أنماط الاتصال السابقة هي أنماط اتصال متعدد إلى واحد أو واحد إلى متعدد، وهو ما يعني ببساطة أن العديد من العمليات ترسل/تستقبل إلى عملية واحدة.

في كثير من الأحيان يكون من المفيد أن تكون قادرًا على إرسال العديد من العناصر إلى العديد من العمليات (أي نمط اتصال متعدد إلى متعدد). MPI_Allgather لديه هذه الخاصية.

نظرًا لمجموعة من العناصر الموزعة عبر جميع العمليات، ستقوم MPI_Allgather بجمع كل العناصر لجميع العمليات. بالمعنى الأساسي، MPI_Allgather هو MPI_Gather متبوعًا بـ MPI_Bcast. كما هو موضح بالشكل

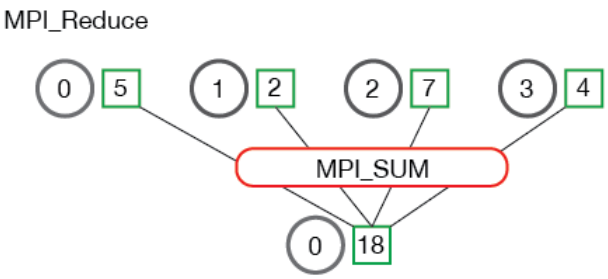


يمتلك الصيغة التالية:

```
MPI_Allgather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    MPI_Comm communicator)
```

MPI_Reduce: على غرار MPI_Gather، يأخذ MPI_Reduce مصفوفة من عناصر الإدخال في كل عملية ويعيد مصفوفة من عناصر الإخراج إلى العملية الجذرية. تحتوي عناصر الإخراج على النتيجة المخفضة. يبدو النموذج الأولي لـ MPI_Reduce كما يلي:

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```



يتضمن هذا التابع مجموعة من العمليات كما هو موضح:

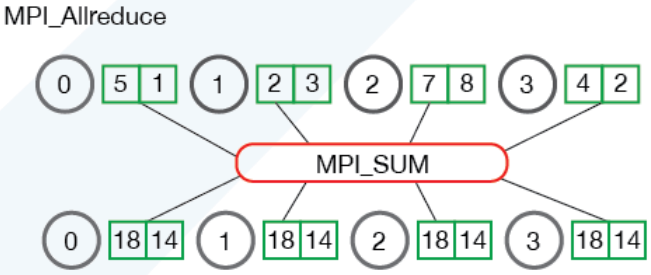
- MPI_MAX - Returns the maximum element.

- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical *and* across the elements.
- MPI_LOR - Performs a logical *or* across the elements.
- MPI_BAND - Performs a bitwise *and* across the bits of the elements.
- MPI_BOR - Performs a bitwise *or* across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

MPI_Allreduce:

تتطلب العديد من التطبيقات المتوازية الوصول إلى النتائج المخفضة عبر جميع العمليات بدلاً من العملية الجذرية. يقوم MPI_Allreduce بتقليل القيم وتوزيع النتائج على كافة العمليات. النموذج الأولي للتابع هو ما يلي:

```
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```



3 تدريب عملي

1.3 تدريب1: مقارنة بين تابع MPI_Bcast الجاهز وآخر مكتوب ضمن البرنامج

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
```

```
        MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);
  if (world_rank == root) {  // If we are the root process, send our data to everyone
   int i;
   for (i = 0; i < world_size; i++) {
    if (i != world_rank) { MPI_Send(data, count, datatype, i, 0, communicator);   }
   }
  } else {  // If we are a receiver process, receive the data from the root
   MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
  }
}
int main(int argc, char** argv) {
  if (argc != 3) {
   fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n");
   exit(1);
  }
  int num_elements = atoi(argv[1]);
  int num_trials = atoi(argv[2]);
  MPI_Init(NULL, NULL);
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  double total_my_bcast_time = 0.0;
  double total_mpi_bcast_time = 0.0;
  int i;
  int* data = (int*)malloc(sizeof(int) * num_elements);
  assert(data != NULL);
  for (i = 0; i < num_trials; i++) {
   // Time my_bcast
   // Synchronize before starting timing
   MPI_Barrier(MPI_COMM_WORLD);
```

```
  total_my_bcast_time -= MPI_Wtime();
  my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
  // Synchronize again before obtaining final time
  MPI_Barrier(MPI_COMM_WORLD);
  total_my_bcast_time += MPI_Wtime();
  // Time MPI_Bcast
  MPI_Barrier(MPI_COMM_WORLD);
  total_mpi_bcast_time -= MPI_Wtime();
  MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
  total_mpi_bcast_time += MPI_Wtime();
 }

 // Print off timing information
 if (world_rank == 0) {
  printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
      num_trials);
  printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);
  printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time / num_trials);
 }
 free(data);
 MPI_Finalize();
}
```

2.3 تدريب2: المطلوب حساب المتوسط الحسابي باستخدام MPI_Scatter and MPI_Gather

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>
// Creates an array of random numbers. Each number has a value from 0 - 1
float *create_rand_nums(int num_elements) {
 float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
 assert(rand_nums != NULL);
```

```
 int i;
 for (i = 0; i < num_elements; i++) {  rand_nums[i] = (rand() / (float)RAND_MAX);  }
 return rand_nums;
}
// Computes the average of an array of numbers
float compute_avg(float *array, int num_elements) {
 float sum = 0.f;
 int i;
 for (i = 0; i < num_elements; i++) {   sum += array[i];  }
 return sum / num_elements;
}
int main(int argc, char** argv) {
 if (argc != 2) {
  fprintf(stderr, "Usage: avg num_elements_per_proc\n");
  exit(1);
 }
 int num_elements_per_proc = atoi(argv[1]);
 // Seed the random number generator to get different results each time
 srand(time(NULL));
 MPI_Init(NULL, NULL);
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int world_size;
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
 // Create a random array of elements on the root process. Its total size will be the number of elements per process times the
number of processes
 float *rand_nums = NULL;
 if (world_rank == 0) {    rand_nums = create_rand_nums(num_elements_per_proc * world_size);  }
 // For each process, create a buffer that will hold a subset of the entire array
 float *sub_rand_nums = (float *)malloc(sizeof(float) * num_elements_per_proc);
 assert(sub_rand_nums != NULL);
 // Scatter the random numbers from the root process to all processes in the MPI world
 MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
         num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);
 // Compute the average of your subset
 float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);
 // Gather all partial averages down to the root process
 float *sub_avgs = NULL;
 if (world_rank == 0) {
```

```
  sub_avgs = (float *)malloc(sizeof(float) * world_size);
  assert(sub_avgs != NULL);
 }
 MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
 if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
  printf("Avg of all elements is %f\n", avg);
  // Compute the average across the original data for comparison
  float original_data_avg =  compute_avg(rand_nums, num_elements_per_proc * world_size);
  printf("Avg computed across original data is %f\n", original_data_avg);
 }
 // Clean up
 if (world_rank == 0) {
  free(rand_nums);
  free(sub_avgs);
 }
 free(sub_rand_nums);
 MPI_Barrier(MPI_COMM_WORLD);
 MPI_Finalize();
}
```

3.3 تدريب3: المطلوب حساب المتوسط الحسابي باستخدام MPI_Allgather

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>
// Creates an array of random numbers. Each number has a value from 0 - 1
float *create_rand_nums(int num_elements) {
 float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
 assert(rand_nums != NULL);
 int i;
 for (i = 0; i < num_elements; i++) {    rand_nums[i] = (rand() / (float)RAND_MAX);  }
 return rand_nums;
}
// Computes the average of an array of numbers
```

```
float compute_avg(float *array, int num_elements) {
 float sum = 0.f;
 int i;
 for (i = 0; i < num_elements; i++) {   sum += array[i];  }
 return sum / num_elements;
}
int main(int argc, char** argv) {
 if (argc != 2) {
  fprintf(stderr, "Usage: avg num_elements_per_proc\n");
  exit(1);
 }
 int num_elements_per_proc = atoi(argv[1]);
 // Seed the random number generator to get different results each time
 srand(time(NULL));
 MPI_Init(NULL, NULL);
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int world_size;
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
 float *rand_nums = NULL;
 if (world_rank == 0) {   rand_nums = create_rand_nums(num_elements_per_proc * world_size);  }
 // For each process, create a buffer that will hold a subset of the entire  array
 float *sub_rand_nums = (float *)malloc(sizeof(float) * num_elements_per_proc);
 assert(sub_rand_nums != NULL);
 // Scatter the random numbers from the root process to all processes in the MPI world
 MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
         num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);
 // Compute the average of your subset
 float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);
 // Gather all partial averages down to all the processes
 float *sub_avgs = (float *)malloc(sizeof(float) * world_size);
 assert(sub_avgs != NULL);
 MPI_Allgather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, MPI_COMM_WORLD);
```

```
 float avg = compute_avg(sub_avgs, world_size);
 printf("Avg of all elements from proc %d is %f\n", world_rank, avg);
 // Clean up
 if (world_rank == 0) {   free(rand_nums); }
 free(sub_avgs);
 free(sub_rand_nums);
 MPI_Barrier(MPI_COMM_WORLD);
 MPI_Finalize();
}
```

4.3 تدريب4: المطلوب إيجاد الأعداد الأولية ضمن مجال أعداد باستخدام البرمجة التفرعية: