Προηγμένες αρχιτεκτονικές υπολογιστών και προγραμματισμός παραλλήλων συστημάτων

LAB-07. MPI – Παραδείγματα και ασκήσεις

Τοπολογία δακτυλίου (1)

- Όλες οι διεργασίες τοποθετούνται πάνω σε ένα νοητό δακτύλιο.
- Κάθε διεργασία επικοινωνεί μόνο με την προηγούμενη και την επόμενη της.
- Λαμβάνει μηνύματα από την προηγούμενη.
- Στέλνει μηνύματα στην επόμενη.

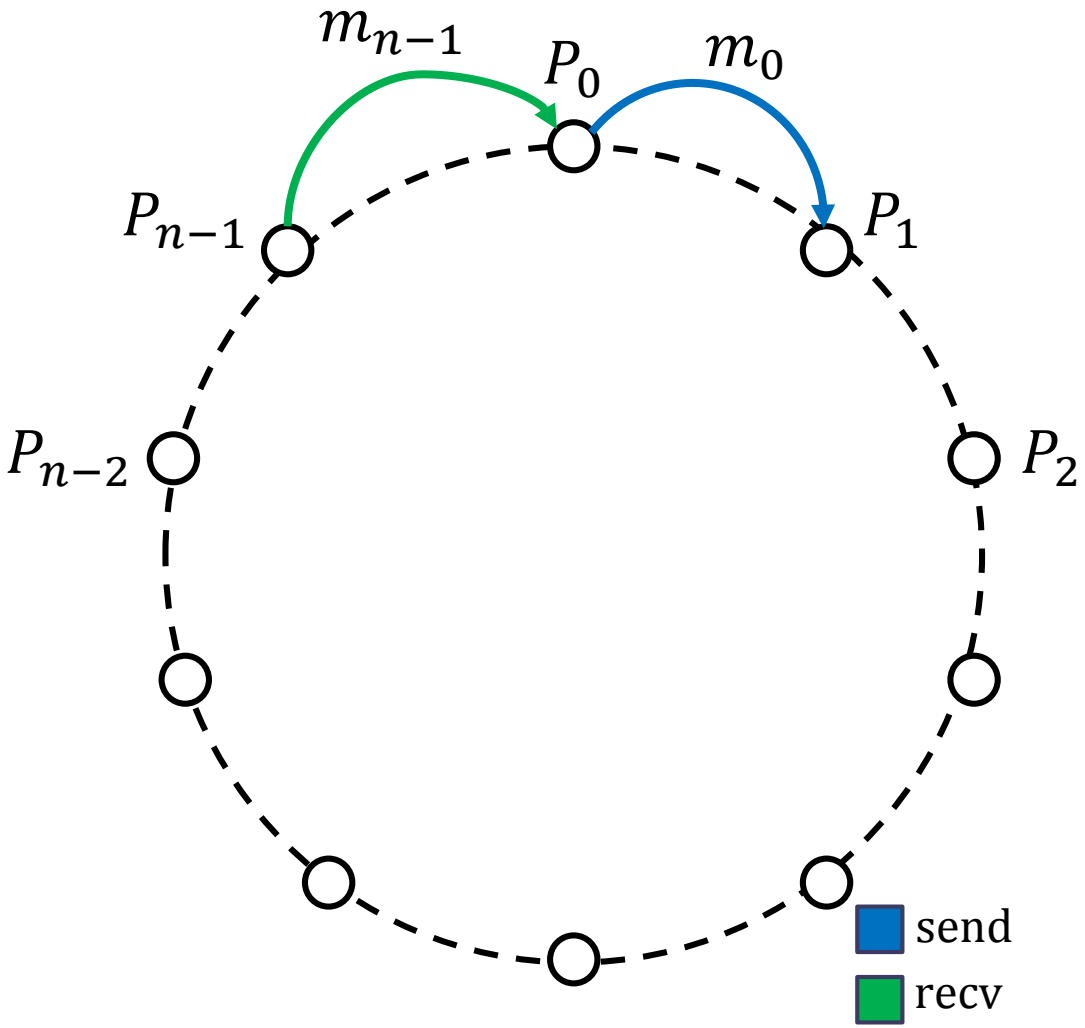
Τοπολογία δακτυλίου n διεργασιών

Η διεργασία P_0:

- Στέλνει μήνυμα m_0 στην P_1: $\text{send}(m_0, P_1)$
- Λαμβάνει μήνυμα m_{n-1} από την P_{n-1}: $\text{recv}(m_{n-1}, P_{n-1})$

Κάθε άλλη διεργασία P_i:

- Λαμβάνει μήνυμα m_{i-1} από την P_{i-1}: $\text{recv}(m_{i-1}, P_{i-1})$
- Στέλνει μήνυμα m_i στην P_{i+1}: $\text{send}(m_i, P_{i+1})$



Υλοποίηση με MPI - Αρχικοποίηση

```
#include "mpi.h"
#include <math.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char **argv) {
     int size, rank, tag = 1, count;
     char msg[100];

     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Υλοποίηση με MPI – Process 0

```
if (rank == 0) {
   /// Construct the message
   sprintf(msg, "Hello from %d", rank);
   /// Send that message to process (1)
   MPI_Send(msg, strlen(msg), MPI_CHAR, 1, tag, MPI_COMM_WORLD);

   /// Receive a message from process (size – 1)
   MPI_Recv(msg,100,MPI_CHAR,size-1, tag, MPI_COMM_WORLD, &status);

   /// Get the number of elements of the previous received message
   MPI_Get_count(&status, MPI_CHAR, &count);
   msg[count] = 0;
   printf("Process %d received \"%s\"\n", rank, msg);
}
```

Υλοποίηση με MPI – Άλλες Processes

```
    else {
        MPI_Recv(msg, 100, MPI_CHAR, rank - 1, tag, MPI_COMM_WORLD,
            &status);
        MPI_Get_count(&status, MPI_CHAR, &count);
        msg[count] = 0;
        printf("Process %d received \"%s\"\n", rank, msg);

        sprintf(msg, "Hello from %d", rank);
        MPI_Send(msg, strlen(msg), MPI_CHAR, (rank + 1), tag,
            MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return(0);
}
```

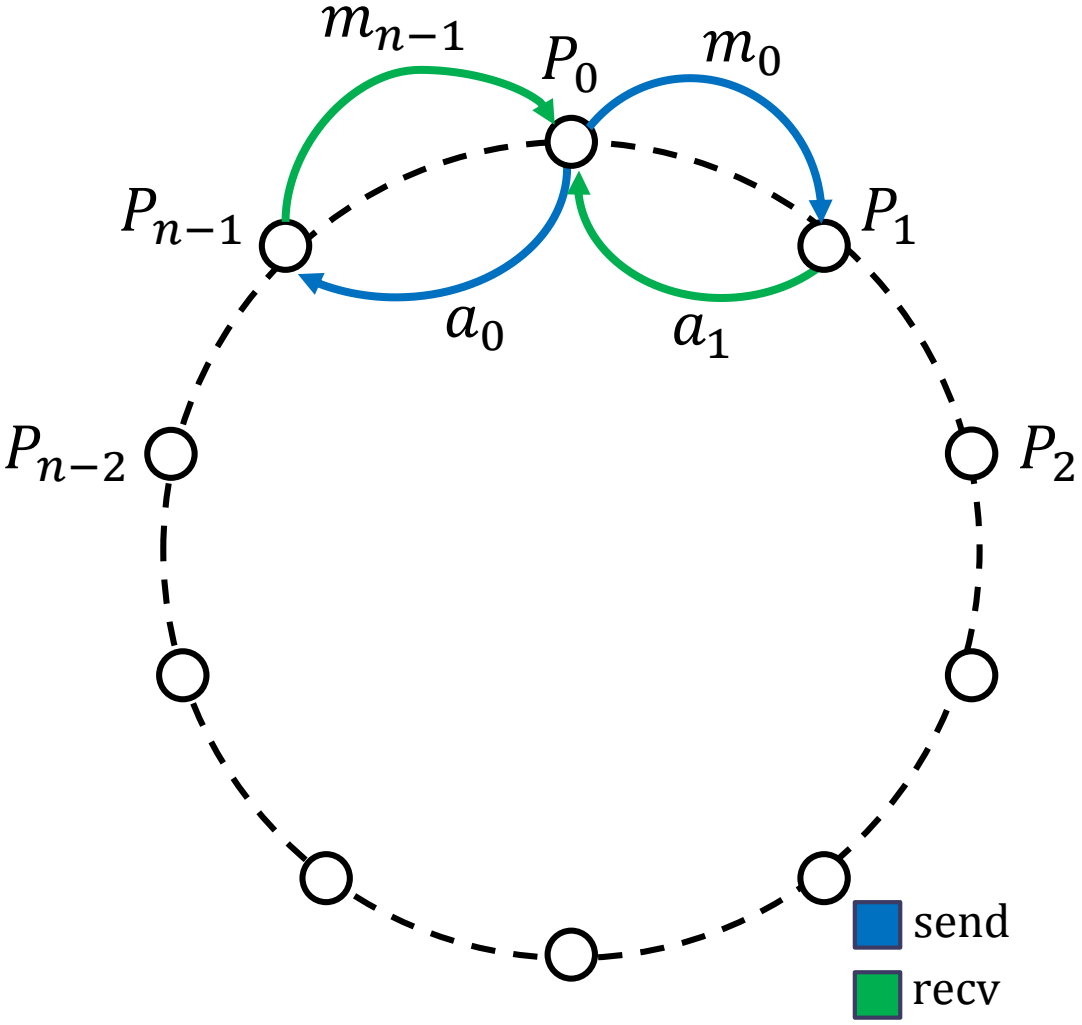
Τοπολογία δακτυλίου με αναγνώριση

- Όλες οι διεργασίες τοποθετούνται πάνω σε ένα νοητό δακτύλιο.
- Κάθε διεργασία επικοινωνεί μόνο με την προηγούμενη και την επόμενη της.
- Λαμβάνει μηνύματα από την προηγούμενη.
  - Στέλνει μήνυμα αναγνώρισης (acknowledgment) στην προηγούμενη ότι το μήνυμα ελήφθη καλώς.
- Στέλνει μηνύματα στην επόμενη.
  - Λαμβάνει μήνυμα αναγνώρισης (acknowledgment) από την επόμενη ότι το μήνυμα εστάλη καλώς.

Τοπολογία δακτυλίου με αναγνώριση

- Όλες οι διεργασίες τοποθετούνται πάνω σε ένα νοητό δακτύλιο.
- Κάθε διεργασία επικοινωνεί μόνο με την προηγούμενη και την επόμενη της.
- Λαμβάνει μηνύματα από την προηγούμενη.
  - Στέλνει μήνυμα αναγνώρισης (acknowledgment) στην προηγούμενη ότι το μήνυμα ελήφθη καλώς.
- Στέλνει μηνύματα στην επόμενη.
  - Λαμβάνει μήνυμα αναγνώρισης (acknowledgment) από την επόμενη ότι το μήνυμα εστάλη καλώς.

Τοπολογία δακτυλίου με αναγνώριση

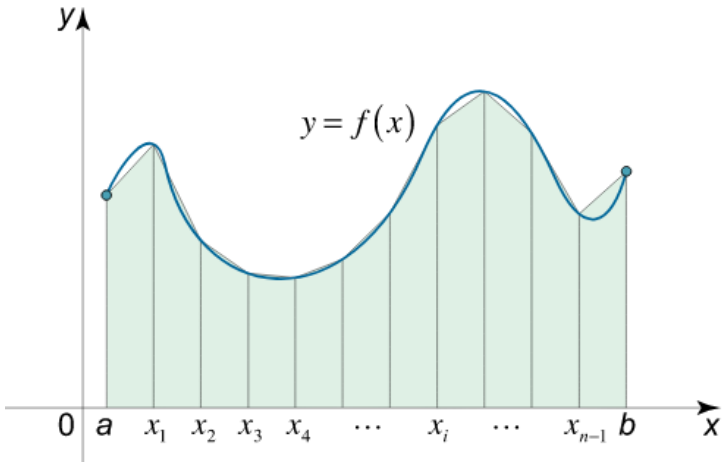


Τοπολογία δακτυλίου με αναγνώριση

- Η διεργασία P_0:
  - Στέλνει μήνυμα m_0 στην P_1: $\text{send}(m_0, P_1)$
  - Λαμβάνει μήνυμα a_1 από την P_1: $\text{recv}(a_1, P_1)$
  - Λαμβάνει μήνυμα m_{n-1} από την P_{n-1}: $\text{recv}(m_{n-1}, P_{n-1})$
  - Στέλνει μήνυμα a_0 στην P_{n-1}: $\text{send}(a_0, P_{n-1})$
- Κάθε άλλη διεργασία P_i:
  - Λαμβάνει μήνυμα m_{i-1} από την P_{i-1}: $\text{recv}(m_{i-1}, P_{i-1})$
  - Στέλνει μήνυμα a_i στην P_{i-1}: $\text{send}(a_i, P_{i-1})$
  - Στέλνει μήνυμα m_i στην P_{i+1}: $\text{send}(m_i, P_{i+1})$
  - Λαμβάνει μήνυμα a_{i+1} από την P_{i+1}: $\text{recv}(a_{i+1}, P_{i+1})$

Τραπεζοειδής ολοκλήρωση συνάρτησης

- Αριθμητική μέθοδος για τον (προσεγγιστικό) υπολογισμό συνάρτησης μιας μεταβλητής.
- Βασίζεται στην ιδιότητα του ότι το ολοκλήρωμα μιας συνάρτησης $f(x)$ μεταξύ των ορίων α και β ισούται με το εμβαδό της επιφάνειας μεταξύ της $y = f(x)$ και του άξονα x (μεταξύ των α και β).



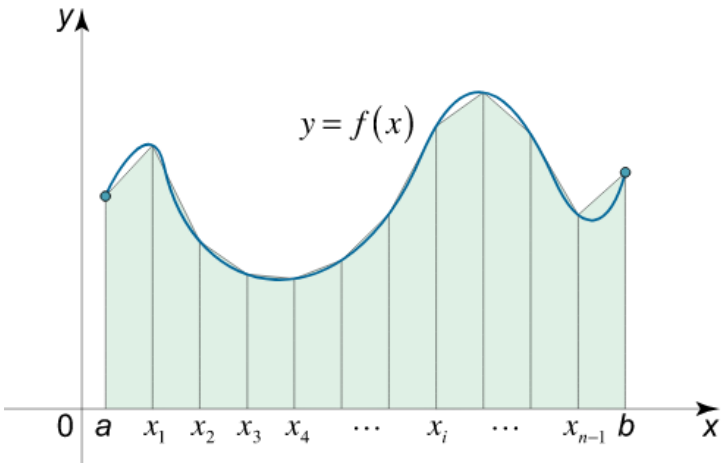
Τραπεζοειδής κανόνας

- Αυτή η επιφάνεια μπορεί να προσεγγιστεί από μία σειρά στοιχειωδών τραπεζίων με ύψος Δx. Δηλαδή:

$$\int_{\alpha}^{\beta} f(x)dx \approx$$

$$\approx \frac{\Delta x}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

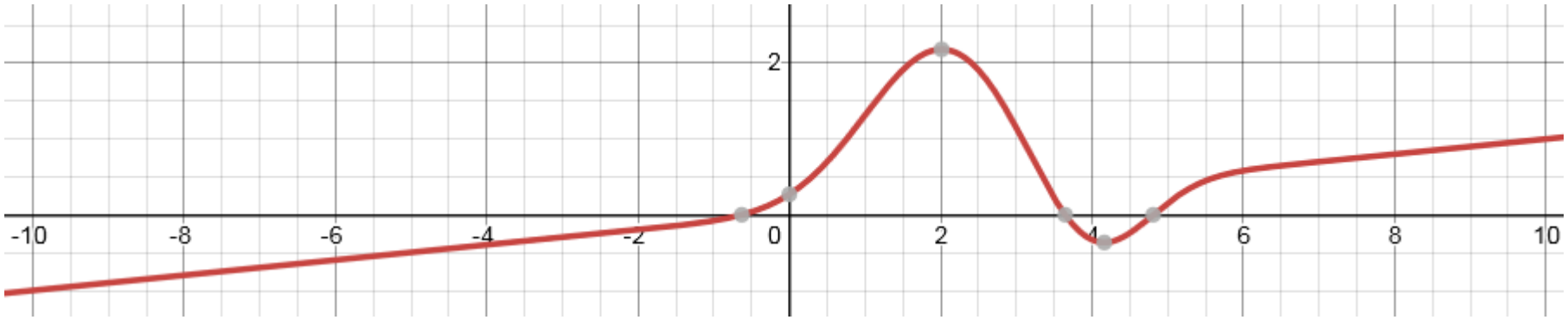
- όπου $\Delta x = (\beta - \alpha)/n$,
- $x_0 = \alpha, x_n = \beta$



Τραπεζοειδής ολοκλήρωση συνάρτησης με το MPI (1)

- Δοθείσας μιας συνάρτησης

$$f(x) = \mathrm{y} = 2e^{-(x-2)^2/2} - e^{-(x-4)^2} + 0.1x$$



Τραπεζοειδής ολοκλήρωση συνάρτησης με το MPI (2)

- Να γραφεί εφαρμογή MPI για τον παράλληλο υπολογισμό του ολοκληρώματος της συνάρτησης με τον προαναφερθέντα κανόνα των τραπεζίων.
- Θα δέχεται ως όρισμα τα όρια α και β καθώς και το πλήθος των διαστημάτων n.
- Κάθε διεργασία θα υπολογίσει και θα προσθέσει εμβαδά διαφορετικών τραπεζίων.

Υλοποίηση με το MPI - Συνάρτηση

```
#include "mpi.h"
#include <math.h>
#include <string.h>
#include <stdio.h>

/// The function to be integrated
double f (double x) {
        double y;
        y = 2 * exp(-(x-2)*(x-2) / 2.0) - exp(-(x-4)*(x-4)) + 0.1*x;

        return y;
}
```

Υλοποίηση με το MPI - Αρχικοποίηση

```
int main(int argc, char **argv) {
    double A = 0, B = 0, ProcA = 0;
    double Height, Integral, ProcIntegral;

    int NumTraps = 0, ProcTraps = 0, NumProcs = 0, ProcRank = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NumProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    if (ProcRank == 0) {
        printf("Parallel trapezoidal Integration with MPI\n\n");

        printf("Please enter the lower bound a (x0): "); fflush(NULL);
        scanf("%lf", &A);

        printf("Please enter the upper bound b (xn): "); fflush(NULL);
        scanf("%lf", &B);

        printf("Please enter the number of trapezoids (n):"); fflush(NULL);
        scanf("%d", &NumTraps);
    }
```

Υλοποίηση με το MPI – Υπολογισμός

```
/// Broadcast A, B, and NumTraps to the rest of the processes
MPI_Bcast(&A, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&B, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&NumTraps, 1, MPI_INT, 0, MPI_COMM_WORLD);

Height = (B - A) / NumTraps;    /// All processes compute the same height

/// Compute the number of trapezoids & the starting point of each process
ProcTraps = ceil((double)NumTraps / NumProcs);
ProcA = A + ProcRank * ProcTraps * Height;

/// For each trapezoid compute its surface and accumulate it.
for (int i = 0; i < ProcTraps; i++) {
   if (ProcA >= B) { break; }           /// We may exceed the B boundary

   ProcIntegral += Height * (f(ProcA) + f(ProcA + Height)) / 2.0;
   ProcA += Height;                      /// Progress the starting point
}
/// Send the local integral values to the root process to add them
MPI_Reduce(&ProcIntegral,&Integral,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

Υλοποίηση με το MPI – Τερματισμός

```
    if (ProcRank == 0) {
        printf("Integral f (%d trapezoids) = %.6f\n", NumTraps, Integral);
    }

    MPI_Finalize();

    return(0);
}
```

Εσωτερικό γινόμενο

- Δοθέντων δύο διανυσμάτων

$$\vec{a} = (a_1, a_2, \dots, a_n)$$
$$\vec{b} = (b_1, b_2, \dots, b_n)$$

- Το εσωτερικό γινόμενο ορίζεται ως το άθροισμα των γινομένων των στοιχείων του διανύσματος.

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^{n} a_i b_i$$

- Να γραφεί εφαρμογή MPI για τον παράλληλο υπολογισμό του εσωτερικού γινομένου δύο διανυσμάτων.

Εσωτερικό γινόμενο με το MPI

- Ανάγνωση του μήκους των διανυσμάτων και των διανυσμάτων από αρχείο.
- Το όνομα του αρχείου μπορεί να περαστεί ως όρισμα στο πρόγραμμα.
- Η root process (rank = 0) θα αναλάβει να διαβάσει τα διανύσματα από το αρχείο, να τα τεμαχίσει και να στείλει σε κάθε process διαφορετικό τμήμα των διανυσμάτων.
- Κάθε process θα αναλάβει το δικό της «μερίδιο» στον υπολογισμό του αθροίσματος.

Εσωτερικό γινόμενο με MPI

```
#include "mpi.h"
#include <math.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char **argv) {
   int NumProcs, ProcRank;

   /// A, B: The vectors to be multiplied.
   /// ProcA, ProcB: Parts of A and B that will be assigned to each process.
   double * A, * B, * ProcA, * ProcB;

   /// vLength: number of elements in A and B
   /// Proc_vLength: number of elements in ProcA and ProcB.
   int i = 0, vLength, Proc_vLength;

   FILE *fid;
   char * sFileName, defaultName[] = "dotprod.dat";

   double Proc_DotProd = 0.0, DotProd = 0.0;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &NumProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
```

```
if (ProcRank == 0) {
   sFileName = argv[1];

   /// Open the file and read the dimensions of the vectors and the vectors.
   fid = fopen(sFileName,"r");

   if (fid) {
      fscanf(fid, "%d", &vLength);      // vector dimensions

      A = (double *)malloc(vLength * sizeof(double));  // alloc mem for vec A
      B = (double *)malloc(vLength * sizeof(double));  // alloc mem for vec B

      for (i = 0; i < vLength; i++)     // Read Vector A
         fscanf(fid, "%lf", &(A[i]));

      for (i = 0; i < vLength; i++)     // Read Vector B
         fscanf(fid, "%lf", &(B[i]));

      fclose(fid);
   } else {
      printf("Input filename %s was not found\n\n", sFileName);
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }
}
```

```
   /// Broadcast n to all other processes
   MPI_Bcast(&vLength, 1, MPI_INT, 0, MPI_COMM_WORLD);

   /// Partition the vectors into NumProcs parts of size ceil(n/ NumProcesses) each.
   Proc_vLength = ceil((double)vLength / (double)NumProcs);
   ProcA = (double *)malloc(Proc_vLength * sizeof(double));
   ProcB = (double *)malloc(Proc_vLength * sizeof(double));

   MPI_Scatter(A,Proc_vLength,MPI_DOUBLE,ProcA,Proc_vLength,MPI_DOUBLE,0, MPI_COMM_WORLD);
   MPI_Scatter(B,Proc_vLength,MPI_DOUBLE,ProcB,Proc_vLength,MPI_DOUBLE,0, MPI_COMM_WORLD);

   /// Compute the assigned dot product, ie. a part of the overall dot product
   Proc_DotProd = 0.0;
   for (i = 0; i < Proc_vLength; i++) {
      Proc_DotProd += ProcA[i] * ProcB[i];
   }

   /// Send the local Products to the root process to be aggregated
   MPI_Reduce(&Proc_DotProd, &DotProd, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   if (ProcRank == 0) {
      printf("Computed Inner Product\t=\t\t%.6f\n", DotProd);
      free(A); free(B);
   }
   free(ProcA); free(ProcB);
   MPI_Finalize();
   return EXIT_SUCCESS;
}
```