

UNIVERSIDAD DEL VALLE DE GUATEMALA

FACULTAD DE INGENIERÍA
CC3069 COMPUTACIÓN PARALELA Y
DISTRIBUIDA

Laboratorio 3

Operaciones vectoriales paralelas utilizando MPI

Adrian Rodriguez, rod21691@uvg.edu.gt
Samuel A. Chamalé, cha21881@uvg.edu.gt

Fecha de entrega: 03/10/2024

Índice

1. Repositorio	1
2. Comunicación Grupal en Funciones de <code>mpi_vector_add.c</code>	1
2.1. <code>Check_for_error()</code>	1
2.2. <code>Read_n()</code>	2
2.3. <code>Read_vector()</code>	3
2.4. <code>Print_vector()</code>	4
2.5. Resumen General	5
3. Modificación de <code>vector_add.c</code> y <code>mpi_vector_add.c</code>	6
4. Medición de Tiempos y Cálculo del Speedup	7
5. Producto Punto y Producto por un Escalar	8
6. Reflexión del Laboratorio	8

1. Repositorio

<https://github.com/chamale-rac/mpi-parallel-vector-operations>

2. Comunicación Grupal en Funciones de `mpi_vector_add.c`

En el programa `mpi_vector_add.c`, se utilizan distintas funciones de comunicación grupal de MPI para coordinar y compartir información entre los procesos que participan en la ejecución paralela. A continuación, se explicará por qué y cómo se usan estos conceptos de forma práctica en las siguientes funciones:

2.1. `Check_for_error()`

Propósito de la Comunicación Grupal:

La función `Check_for_error()` se utiliza para verificar si algún proceso ha encontrado algún error durante la ejecución. Es importante que todos los procesos estén al tanto del estado de fallo de los demás para poder manejar el error de manera coherente y evitar que la orquestación comience a presentar inconsistencias.

Uso de Comunicación Grupal:

La función utiliza `MPI_Allreduce` para realizar una operación de reducción que combinará los valores de una variable local (`local_ok`) de todos los procesos para producir un resultado global (`ok`). Siendo específicos, se utiliza la operación `MPI_MIN` para determinar si algún proceso ha establecido `local_ok` en 0 (indicando un error).

Fragmento de Código:

```
void Check_for_error(int local_ok, char fname[], char message[], MPI_Comm
comm) {
    int ok;
    MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
    if (ok == 0) {
        int my_rank;
        MPI_Comm_rank(comm, &my_rank);
        if (my_rank == 0) {
            fprintf(stderr, "Proc %d > En %s, %s\n", my_rank, fname, message);
            fflush(stderr);
        }
        MPI_Finalize();
        exit(-1);
    }
}
```

Explicación Detallada:

- `MPI_Allreduce`: Todos los procesos llaman a esta función para compartir su valor de `local_ok`.
- La operación `MPI_MIN` asegura que si algún proceso tiene `local_ok` igual a 0, el valor global `ok` será 0.
- Si `ok` es 0, significa que al menos un proceso encontró un error, y por lo tanto, todos los procesos pueden tomar medidas apropiadas, como finalizar la ejecución.

2.2. `Read_n()`

Propósito de la Comunicación Grupal:

La función `Read_n()` se encarga como tal de leer el tamaño (`n`) desde la entrada estándar en el proceso *root* (normalmente el proceso 0) para luego comunicar este valor a todos los demás procesos y asegurar que todos trabajan con el mismo tamaño de vector.

Uso de Comunicación Grupal:

Se utiliza `MPI_Bcast` para difundir el valor de `n` desde el proceso raíz a todos los demás procesos en el comunicador.

Fragmento de Código:

```
void Read_n(int* n_p, int* local_n_p, int my_rank, int comm_sz, MPI_Comm comm)
{
    int local_ok = 1;
    char *fname = "Read_n";

    if (my_rank == 0) {
        printf("What's the order of the vectors?\n");
        scanf("%d", n_p);
    }
    MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
    if (*n_p <= 0 || *n_p % comm_sz != 0) local_ok = 0;
    Check_for_error(local_ok, fname,
        "n should be > 0 and evenly divisible by comm_sz", comm);
    *local_n_p = *n_p/comm_sz;
}
```

Explicación Detallada:

- `MPI_Bcast`: El proceso raíz envía el valor de `n` a todos los demás procesos.
- Todos los procesos, incluyendo el raíz, llaman a `MPI_Bcast`, asegurando que todos reciban el mismo valor de `n`.
- Esto es esencial para que cada proceso pueda calcular su parte local del vector (`local_n`).

2.3. `Read_vector()`

Propósito de la Comunicación Grupal:

La función `Read_vector()` tiene como propósito principal leer el vector completo en el proceso *root* y distribuir equitativamente sus partes a todos los procesos participantes.

Uso de Comunicación Grupal:

Se utiliza `MPI_Scatter` para dividir el vector global en porciones y enviar cada porción a los procesos correspondientes.

Fragmento de Código:

```
void Read_vector(double local_a[], int local_n, int n, char vec_name[], int
    my_rank, MPI_Comm comm) {
    double* a = NULL;
    int i;
    int local_ok = 1;
    char* fname = "Read_vector";
    if (my_rank == 0) {
        a = malloc(n * sizeof(double));
        if (a == NULL) local_ok = 0;
        Check_for_error(local_ok, fname, "No se puede asignar vector
            temporal", comm);
        // Inicializacion del vector 'a'
        for (i = 0; i < n; i++)
            a[i] = i;
    }
    MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
    if (my_rank == 0) free(a);
}
```

Explicación Detallada:

- `MPI_Scatter`: El proceso raíz divide el vector global `a` y envía una porción de `local_n` elementos a cada proceso.
- Cada proceso recibe su porción correspondiente en `local_a`.
- Esto permite que cada proceso trabaje en paralelo con su parte del vector.

2.4. Print_vector()

Propósito de la Comunicación Grupal:

La función `Print_vector()` se encarga de reunir las porciones locales de un vector distribuidas entre los procesos y ensamblarlas en el proceso raíz para imprimir el vector completo.

Uso de Comunicación Grupal:

Se utiliza `MPI_Gather` para recopilar las partes locales de cada proceso y reunir las en un vector global en el proceso *root*.

Fragmento de Código:

```
void Print_vector(double local_b[], int local_n, int n, char title[], int
    my_rank, MPI_Comm comm) {
    double* b = NULL;
    int i;
    int local_ok = 1;
    char* fname = "Print_vector";

    if (my_rank == 0) {
        b = malloc(n*sizeof(double));
        if (b == NULL) local_ok = 0;
        Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++)
            printf("%f ", b[i]);
        printf("\n");
        free(b);
    } else {
        Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0,
            comm);
    }
}
```

Explicación Detallada:

- **MPI_Gather:** Cada proceso envía su vector local `local_b` al proceso raíz.
- El proceso raíz recibe todas las porciones y las almacena en el vector global `b`.
- Una vez reunido el vector completo, el proceso raíz puede imprimirlo o realizar otras operaciones.

Nota Adicional:

- Es importante que todos los procesos llamen a **MPI_Gather**, incluso si solo el proceso raíz necesita el vector completo. Esto asegura la sincronización y correcta ejecución del programa.

2.5. Resumen General

La comunicación grupal en estas funciones es esencial para:

- Garantizar la coherencia de datos entre procesos.
- Dividir y distribuir el trabajo de manera eficiente.
- Recolectar y consolidar resultados parciales.
- Manejar errores de forma coordinada.

El uso de funciones colectivas como **MPI_Bcast**, **MPI_Scatter**, **MPI_Gather** y **MPI_Allreduce** facilita la implementación de algoritmos paralelos en el contexto de MPI y optimiza el rendimiento al reducir la necesidad de comunicaciones punto a punto más complejas.

3. Modificación de `vector_add.c` y `mpi_vector_add.c`

Para este apartado, se modificaron los programas `vector_add.c` y `mpi_vector_add.c` para crear dos vectores de al menos 100,000 elementos generados de forma aleatoria. Las modificaciones incluyeron:

- Lectura del tamaño del vector (`n`) desde los argumentos de la ejecución.
- Generación de vectores de números aleatorios en lugar de la lectura desde la entrada estándar.
- Impresión únicamente de los primeros y últimos 10 elementos de los vectores para validación.
- Medición del tiempo de ejecución para la suma de los vectores.

A continuación, se presentan capturas de pantalla de los resultados obtenidos al ejecutar los programas:

```
➡ schr mpi-parallel-vector-operations X ./vector_add 1000000
=> The first vector is
    0.727222 0.234743 0.201154 0.957320 0.871293 0.262287 0.447019 0.218731 0.038059 0.857902
    ...
    0.996146 0.365869 0.330565 0.929035 0.946525 0.438162 0.883330 0.828391 0.005219 0.839849
=> The second vector is
    0.084785 0.647245 0.004862 0.276611 0.804482 0.484353 0.307262 0.012590 0.344976 0.941889
    ...
    0.414629 0.802367 0.681801 0.267839 0.399038 0.169554 0.868036 0.259095 0.347002 0.573806
=> The sum is
    0.812007 0.881989 0.206016 1.233931 1.675774 0.746640 0.754281 0.231320 0.383035 1.799791
    ...
    1.410775 1.168236 1.012366 1.196874 1.345563 0.607716 1.751366 1.087485 0.352221 1.413655
Vector addition took 0.006778 seconds
```

Figura 1: Salida de la ejecución del programa `vector_add.c` con 100,000 elementos.

```
➡ schr mpi-parallel-vector-operations X mpiexec -n 4 ./mpi_vector_add 100000
=> The first vector is
    0.145627 0.387492 0.331643 0.961954 0.071599 0.301235 0.997038 0.468709 0.810158 0.590266
    ...
    0.592386 0.649541 0.047069 0.392072 0.021350 0.343001 0.330829 0.721897 0.794695 0.904281
=> The second vector is
    0.002803 0.299486 0.633338 0.780643 0.504307 0.520934 0.356390 0.262123 0.114317 0.672984
    ...
    0.590673 0.565791 0.718339 0.928664 0.373375 0.982873 0.750197 0.893129 0.521700 0.772342
=> The sum is
    0.148430 0.686978 0.964981 1.742598 0.575905 0.822170 1.353429 0.730832 0.924475 1.263250
    ...
    1.183058 1.215333 0.765409 1.320736 0.394725 1.325874 1.081026 1.615026 1.316394 1.676624
Vector addition took 0.000622 seconds
```

Figura 2: Salida de la ejecución del programa `mpi_vector_add.c` con 100,000 elementos usando MPI.

4. Medición de Tiempos y Cálculo del Speedup

Para esta sección, se midieron los tiempos de ejecución de ambos programas (`vector_add.c` y `mpi_vector_add.c`) con el fin de calcular el *speedup* logrado mediante la versión paralela. Las mediciones se realizaron utilizando un script automatizado en `bash`, el cual ejecuta ambos programas 10 veces, extrae los tiempos de ejecución, y calcula el promedio para cada uno. Posteriormente, utiliza estos promedios para calcular el *speedup*.

Debido a las limitaciones de recursos del sistema, el tamaño máximo de los vectores que se pudo utilizar fue de 200,000,000 elementos.

A continuación, se presenta una captura de pantalla con los resultados obtenidos:

```
➔ schr mpi-parallel-vector-operations X ./measure_speedup.sh
Running sequential program (./vector_add)...
Sequential run 1: 2.625234 seconds
Sequential run 2: 1.716244 seconds
Sequential run 3: 1.288468 seconds
Sequential run 4: 1.734576 seconds
Sequential run 5: 1.476415 seconds
Sequential run 6: 1.388742 seconds
Sequential run 7: 1.124459 seconds
Sequential run 8: 1.210090 seconds
Sequential run 9: 1.409647 seconds
Sequential run 10: 1.383043 seconds
Running parallel program (mpiexec -n 4 ./mpi_vector_add)...
Parallel run 1: 0.442052 seconds
Parallel run 2: 0.410237 seconds
Parallel run 3: 0.375187 seconds
Parallel run 4: 0.495756 seconds
Parallel run 5: 0.415867 seconds
Parallel run 6: 0.411623 seconds
Parallel run 7: 0.393464 seconds
Parallel run 8: 0.387288 seconds
Parallel run 9: 0.391083 seconds
Parallel run 10: 0.420583 seconds

-----
| Program          | Average Time (seconds) |
|-----|-----|
| Sequential       | 1.535691                |
| Parallel (4)     | .414314                 |
|-----|-----|

Speedup: 3.706587
```

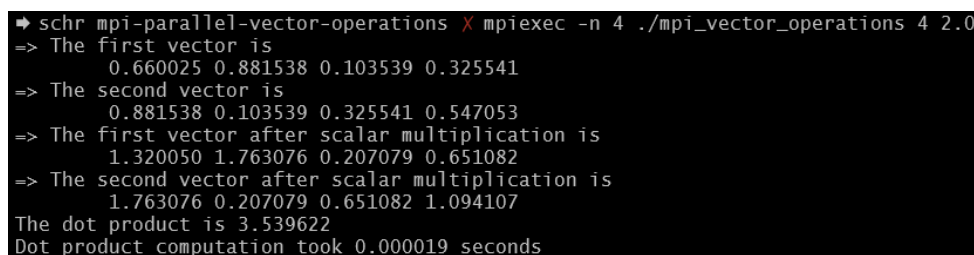
Figura 3: Resultados de la ejecución automatizada mostrando los tiempos y el cálculo del *speedup*.

5. Producto Punto y Producto por un Escalar

Se ha modificado el programa original `mpi_vector_add.c` para realizar dos operaciones en paralelo sobre dos vectores:

1. **Producto Punto:** Se calcula el producto punto de dos vectores distribuidos entre los procesos. Cada proceso calcula el producto punto de su porción local y, posteriormente, se utiliza `MPI_Reduce` para sumar los resultados locales y obtener el producto punto global.
2. **Producto de un Escalar por Cada Vector:** Se implementa la multiplicación de cada vector por un escalar en paralelo. Cada proceso multiplica su porción local de los vectores por el escalar especificado, utilizando una nueva función `Parallel_scalar_multiplication`.

A continuación, se presente una captura de pantalla con los resultados obtenidos:



```
➡ schr mpi-parallel-vector-operations X mpiexec -n 4 ./mpi_vector_operations 4 2.0
=> The first vector is
    0.660025 0.881538 0.103539 0.325541
=> The second vector is
    0.881538 0.103539 0.325541 0.547053
=> The first vector after scalar multiplication is
    1.320050 1.763076 0.207079 0.651082
=> The second vector after scalar multiplication is
    1.763076 0.207079 0.651082 1.094107
The dot product is 3.539622
Dot product computation took 0.000019 seconds
```

Figura 4: Resultados de las operaciones de Producto.

6. Reflexión del Laboratorio

A lo largo de este laboratorio, aplicamos distintas técnicas de programación paralela con MPI para mejorar el rendimiento de operaciones vectoriales. Los elementos clave implementados fueron la distribución de entre procesos, la sincronización de las operaciones y la agregación de resultados (`MPI_Barrier` y `MPI_Reduce`).

Como mejoras futuras, se podría considerar la optimización adicional de la generación de números aleatorios para reducir la sobrecarga en la inicialización de datos. También se podrían realizar pruebas con un mayor número de procesos y vectores de gran tamaño para analizar el escalado del programa.

Referencias

- [Mpi] *Using MPI with C — Research Computing, University of Colorado Boulder documentation*. <https://curc.readthedocs.io/en/latest/programming/MPI-C.html>. Accessed: 2024-10-03. n.d.

Índice de figuras

1. Salida de la ejecución del programa `vector_add.c` con 100,000 elementos. . 6
2. Salida de la ejecución del programa `mpi_vector_add.c` con 100,000 elementos usando MPI. 6
3. Resultados de la ejecución automatizada mostrando los tiempos y el cálculo del *speedup*. 7
4. Resultados de las operaciones de Producto. 8