Universidad Nacional de La Plata

Facultad virtual de Ingeniería e Informática

Entrega Nº2

Programación con memoria distribuida

Sistemas distribuidos y paralelos

12 de mayo de 2018

Andreini, Antonella 822/9

Flores, Brian 795/4

El código de los ejercicios se encuentra disponible en
https://github.com/brianflores/sdp-entrega2

1. Realizar un algoritmo MPI que resuelva la expresión:

$$M = \overline{u.l.}(AB + LC + DU)$$

Donde A, B, C y D son matrices de NxN. L y U son matrices triangulares de NxN inferior y superior, respectivamente. $\bar{u}$ y $\bar{l}$ son los promedios de los valores de los elementos de la matrices U y L, respectivamente.
Evaluar N=512, 1024 y 2048

Resolución con un algoritmo secuencial:

En este ejercicio se realizaron las siguientes operaciones con matrices:

1. Declaración
2. Cálculo de los promedios
3. Multiplicación de matrices
4. Check del resultado
5. Liberación de memoria

En primer lugar se reserva memoria para todas las matrices y se inicializan. Hay una excepción con las matrices triangulares, como L queda del lado derecho de la multiplicación es necesario que se inicializada totalmente pero U queda del lado izquierdo entonces se descartan los valores que son 0 y sólo se inicializan con los unos.

```
int elementosU = (N*N)-((N*(N-1))/2);
for(i=0; i<elementosU; i++) U[i] = 1.0;
```

Los elementos de U fueron calculados para que la matriz quede inicializada como triangular superior.

Luego, se realizó la suma de todos los elementos para dividirlo por el total y asi calcular los promedio de u y l

```
unsigned long Total = N*N;
for(i=0;i<N;i++){   //Calcula los promedios
   for(j=0;j<N;j++){
       promedioL+= L[i*N+j];
       //promedioU+= U[i*N+j];
   }
}
for(i=0; i<elementosU; i++) promedioU+= U[i];
promedioL = promedioL / Total;
promedioU = promedioU / Total;
promedioL = promedioL * promedioU; //En promedioL queda el promedio de L por el de U.
```

Después se empieza a descomponer el cálculo de la matriz M en partes, empezando por reservar memoria para la matriz AB y realizando su multiplicación. En este punto es bueno aclarar que se podrían haber realizado varias multiplicaciones de matrices dentro de un mismo *for*, pero esto causaría un aumento en los fallos de caché y bajaría el rendimiento del programa. Por lo tanto, cada multiplicación de matrices se lleva a cabo de manera individual.

```
AB=(double*)malloc(sizeof(double)*N*N); //AB=A*B
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        AB[i*N+j]=0;
        for(k=0;k<N;k++){
                AB[i*N+j]= AB[i*N+j] + A[i*N+k]*B[k+j*N];
        }
    }
}
```

La multiplicación se realiza por columnas por haber quedado comprobado en las anteriores prácticas una mejor eficiencia en los tiempos de ejecución.

Las siguientes matrices a calcular son LC y DU que se realiza de forma similar que AB, con la salvedad de que L y U son matrices triangulares por lo que solo es necesario recorrer los elementos no nulos de las mismas

```
LC=(double*)malloc(sizeof(double)*N*N); //LC= L*C
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        LC[i*N+j]=0;
        for(k=i;k<N;k++){
                LC[i*N+j]= LC[i*N+j] + L[i*N+k]*C[k+j*N];
        }
    }
}
```

```
DU=(double*)malloc(sizeof(double)*N*N); //DU= D*U
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        DU[i*N+j]=0;
        for(k=0;k<=j;k++){
                DU[i*N+j]= DU[i*N+j] + D[i*N+k]*U[k+j*(j+1)/2];
        }
    }
}
```

Finalmente, una vez obtenidas las matrices de cada término, se realiza la suma de las mismas y luego la multiplicación por el promedio l que ya estaba multiplicado por el promedio u.

```
TOTAL=(double*)malloc(sizeof(double)*N*N); //TOTAL= (AB+LC+DU)*promedioL
for(i=0;i<N;i++){
    for(j=0;j<N;j++)
    TOTAL[i*N+j]= (AB[i*N+j] + LC[i*N+j] + DU[i*N+j])*promedioL;
}
```

Para realizar el check del resultado de las matrices se calculó un ejemplo utilizando matrices de 4x4,sin multiplicar por el promedio y el resultado era

```
9.000000 10.000000 11.000000 12.000000
8.000000 9.000000 10.000000 11.000000
7.000000 8.000000 9.000000 10.000000
6.000000 7.000000 8.000000 9.000000
```

Por lo tanto se desarrolló un algoritmo que recorra cada fila viendo que el valor de cada elemento sea igual al del elemento anterior + 1 y que luego recorra cada columna

comprobando que el valor de cada elemento sea igual al del elemento anterior - 1.

```
    double resultado;
    for(i=0;i<N;i++){
            resultado = TOTAL[i*N]/promedioL;
    for(j=0;j<N;j++){
    check = check && (TOTAL[i*N+j]/promedioL==resultado+j);
    }
    }
    for(j=0;j<N;j++){
            resultado = TOTAL[j*N]/promedioL;
            for(i=0;i<N;i++){
                check = check && (((TOTAL[i+j*N]/promedioL)-resultado-i)==0);
    }
    }
if(check){
    printf("Multiplicacion de matriz correcta\n");
}else{
    printf("Multiplicacion de matriz erroneo\n");
}
```

Para probar este algoritmo, ejecutar `./ejercicio1secuencial N`, siendo N la cantidad de filas y columnas de las matrices.

Resolución con un algoritmo paralelizado con MPI:

En primer lugar para desarrollar un algoritmo en MPI se inicializan las comunicaciones:

```
MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &ID);
    MPI_Comm_size(MPI_COMM_WORLD, &cantProcesos);
if (ID == 0) root(N,cantProcesos);
else if (ID > 0) workers(ID,N,cantProcesos);
MPI_Finalize();
```

Los comandos utilizados significan lo siguiente:

- Mpi_Init: Inicializa el ambiente de ejecución de MPI y recibe como parámetros los argumentos del programa.
- Mpi_Comm_rank: permite obtener el identificador del proceso, y recibe como parámetros los ID de los procesos y MPI_COMM_WORLD que es el comunicador por defecto que incluye a todos los procesos en la ejecución.
- MPI_Comm_size: permite obtener la cantidad de procesos creados.
- MPI_Finalize: termina la ejecución del ambiente MPI.

Además, dentro del ambiente de MPI se le asigna al proceso de id igual a 0 como root y a los demás como workers.

Proceso root

Este proceso se simplifica en los siguientes pasos;

- Inicialización de las matrices
- Comunicaciones
- Cálculo de las matrices
- Comunicaciones finales

Al principio el proceso root reserva lugar en la memoria para las matrices y las inicializa, para dividir el trabajo que realizará cada proceso y enviarle una fracción de matriz a cada worker.

```
int filas = N/cantProcesos; //filas por proceso
MPI_Scatter(A, N*filas, MPI_DOUBLE, a, N*filas, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(B,N*N, MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Scatter(L, N*filas, MPI_DOUBLE, l, N*filas, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(C,N*N, MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Scatter(D, N*filas, MPI_DOUBLE, d, N*filas, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(U,elementosU, MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Scatter(TOTAL, N*filas, MPI_DOUBLE, total, N*filas, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

El comando MPI_Scatter es el encargado de enviar un mensaje dividido en partes iguales al resto de procesos y a sí mismo y MPI_Bcast envía un mensaje desde un proceso origen a todos los procesos. La diferencia se encuentra en que las matrices ubicadas a la izquierda en la multiplicación van a dividirse en partes para todos los procesos y las matrices ubicadas a la derecha de la multiplicación se envían completas.

Después, se calcula la fracción de promedios que le corresponde y se realiza una comunicación bloqueante MPI_Allreduce hasta que todos los procesos calculen su fracción de promedio y reciban el resultado de los demás.

```
MPI_Allreduce(&promedioL, &resultadoL, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&promedioU, &resultadoU, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Una vez obtenido el resultado de los dos promedios, se dividen por el total y deja en el promedio L, la multiplicación de ambos promedios.

Después se realiza el cálculo de la fracción de multiplicación de matrices que le corresponde y espera el resultado de los demás procesos y los almacena en una matriz denominada TOTAL.

```
MPI_Gather(total, filas*N, MPI_DOUBLE, TOTAL, filas*N, MPI_DOUBLE, 0, MPI_COMM_WORLD); //Cada proceso envia su pedacito de matriz,
```

Para finalizar verifica si el resultado es correcto con el check.

Procesos workers

Estos procesos comienzan reservando memoria para cada copia de matriz que fue enviada por el proceso root utilizando el comando Bcast. Luego, reciben todas las comunicaciones del root y calculan la fracción de promedio que les corresponde hasta quedarse bloqueados en el MPI_Allreduce esperando el resultado de los promedios de todos los procesos.

Después de recibir el resultado de los promedios, los divide por el total y deja la multiplicación de ambos promedios en el promedio l.

Por último, calcula la fracción de la multiplicación de matrices que les corresponde y le envía al root el resultado.

Para probar este algoritmo, ejecutar:

- En una máquina con cuatro procesos: `mpirun -np 4 ejercicio1mpi 512`
- En dos máquinas:
  - `mpirun -np 4 --machinefile maquinasej1_4procesos_2maquinas ejercicio1mpi 512` (dos máquinas con 2 procesos cada una)
  - `mpirun -np 4 --machinefile maquinasej1_8procesos ejercicio1mpi 512` (dos máquinas con 4 procesos cada una)

Tiempos de los algoritmos:

N	Secuencial	MPI (una máquina, 4 procesos)	MPI (dos máquinas, 4 procesos)	MPI (dos máquinas, 8 procesos)
512	Ts = 1.533326	Troot = 0.493336 Tworker1 = 0.481700 Tworker2 = 0.485840 Tworker3 = 0.483825	Troot = 0.575663 Tworker1 = 0.459306 Tworker2 = 0.451456 Tworker3 = 0.447904	Troot = 0.471316 Tworker1: 0.261722 Tworker2: 0.259672 Tworker3: 0.253435 Tworker4: 0.248955 Tworker5: 0.246469 Tworker6: 0.245093 Tworker7: 0.243482
1024	Ts = 12.087812	Troot = 3.861348 Tworker1 = 3.826384 Tworker2 = 3.811224 Tworker3 = 3.812694	Troot = 4.088227 Tworker1 = 3.710428 Tworker2 = 3.683137 Tworker3 = 3.665770	Troot = 2.377366 Tworker 5: 1.960309 Tworker 6: 1.953343 Tworker 7: 1.944580 Tworker 4: 1.972119 Tworker 3: 1.986927 Tworker 1: 1.992528 Tworker 2: 1.990282
2048	Ts = 97.188568	Troot = 30.613317 Tworker1 = 30.353058 Tworker2 = 30.326442 Tworker3 = 30.321027	Troot = 31.031770 Tworker1 = 29.417363 Tworker2 = 29.267310 Tworker3 = 29.191078	Troot = 17.173323 Tworker 6: 15.512670 Tworker 4: 15.587824 Tworker 5: 15.558857 Tworker 7: 15.489087 Tworker 2: 15.698976 Tworker 3: 15.713172 Tworker 1: 15.780363

Speedup y eficiencia de los algoritmos respecto al secuencial:

N	MPI (una máquina, 4 procesos)	MPI (dos máquinas, 4 procesos)	MPI (dos máquinas, 8 procesos)
512	S = 3.1080 E = 0.777	S = 2.6635 E = 0.6658	S = 3.2532 E = 0.4066
1024	S = 3.1304 E = 0.7826	S = 2.9567 E = 0.7391	S = 5.0845 E = 0.6355

2048	S = 3.1747 E = 0.7936	S = 3.1319 E = 0.7829	S = 5.6592 E = 0.7074

Análisis de escalabilidad

Comm_sz	N = 512	N = 1024	N = 2048
4 (una máquina)	E = 0.777	E = 0.7826	E = 0.7936
4 (dos máquinas)	E = 0.6658	E = 0.7391	E = 0.7829
8	E = 0.4066	E = 0.6355	E = 0.7074

El programa no es fuertemente escalable, ya que al aumentar el número de procesos dejando el tamaño del problema constante, la eficiencia decae.

Es débilmente escalable porque la eficiencia se mantiene relativamente constante (entre 0.7 y 0.8) al aumentar el número de procesos y el tamaño del problema.

Balance de cargas:

N	MPI (una máquina, 4 procesos)	MPI (dos máquinas, 4 procesos)	MPI (dos máquinas, 8 procesos)
512	B=0.985485	B=0.840044	B=0.591446
1024	B=0.991334	B=0.926291	B=0.850597
2048	B=0.99314494	B=0.9579498769	B=0.9208

En general se obtuvo un buen balance de cargas, ya que todos los procesos trabajan con porciones iguales de cada matriz, excepto en el caso de la matriz U, que no puede ser distribuida en partes iguales.

Impacto de las comunicaciones:

N	MPI (una máquina, 4 procesos)	MPI (dos máquinas, 4 procesos)	MPI (dos máquinas, 8 procesos)
512	Tcomunicaciones = 0.00985	Tcomunicaciones = 0.121622	Tcomunicaciones = 0.226244
1024	Tcomunicaciones = 0.028054	Tcomunicaciones = 0.390359	Tcomunicaciones = 0.403525
2048	Tcomunicaciones = 0.085101	Tcomunicaciones = 1.567162	Tcomunicaciones = 1.567923

2. Para el ejercicio anterior realizar un algoritmo híbrido MPI-OpenMP y comparar con los resultados obtenidos anteriormente.

Para realizar el algoritmo híbrido se tomó el código del algoritmo de MPI y se cambió lo siguiente:

- Se agregó en el main la cantidad de threads

```
omp_set_num_threads(4);
```

- Se calcula el promedio paralelizando con pragma, agregando un collapse 2 para que paralelizar los dos for anidados y por último utiliza reduction para sumar el resultado de los 4 threads.

```
promedioL = 0;
promedioU = 0;
#pragma omp parallel for collapse(2) reduction(+:promedioL,promedioU)
    for(i=0;i<filas;i++){   //Calcula los promedios
        for(j=0;j<N;j++){
            promedioL+= l[i*N+j];
            //promedioU+= U[i*N+j];
        }
    }
#pragma omp parallel for reduction(+:promedioU)
    for(i=0; i<elementosUporProceso; i++) promedioU+= U[i];
```

- Por último, a cada multiplicación de matrices se le agrega el pragma y se le asigna a cada thread una variable k privada

```
#pragma omp for collapse(2) private(k)
for(i=0;i<filas;i++){   //AB = a*B
    for(j=0;j<N;j++){
            AB[i*N+j]=0;
            for(k=0;k<N;k++){
                AB[i*N+j]= AB[i*N+j] + a[i*N+k]*B[k+j*N];
            }
    }
}
```

Para probar este algoritmo, ejecutar: `mpirun -np 2 --machinefile maquinas_ej2 ejercicio2 512` (dos máquinas con 4 threads cada una).

Tiempo de algoritmo:

N	Secuencial	Híbrido (dos máquinas, con 4 threads cada una)
512	Ts = 1.533326	Troot = 1.173977 Tworker1=1.063715
1024	Ts = 12.087812	Troot =9.034866 Tworker1=8.646166
2048	Ts = 97.188568	Troot = 70.209813 Tworker1=68.690816

Aclaración: los tiempos obtenidos son muy grandes producto de un error en la versión de MPI instalada en las computadoras usadas para probar el programa. El error provocaba que los 4 threads de cada proceso se ejecutaran en un mismo core, por lo que estimamos que los tiempos correctos serían aproximadamente 4 veces menores a los obtenidos.
Lamentablemente descubrimos la causa del error luego de la última fecha de pruebas de tiempos, por lo que pudimos correr el programa en otras computadoras.

Speedup y eficiencia del algoritmo con respecto al secuencial:

N	Híbrido (dos máquinas, con 4 threads cada una)
512	S = 1.3060 E = 0.16325
1024	S = 1.33790717 E = 0.173032
2048	S = 1.384259 E = 0.1730323

A modo de conclusión podemos decir que los algoritmos MPI puros son más eficientes en términos de tiempos de ejecución, pero los algoritmos híbridos cuentan con la ventaja de que consumen menos memoria, ya que los threads de cada máquina pueden trabajar con memoria compartida.

Análisis de escalabilidad:

En este caso no podemos hacer un análisis completo, ya que no tenemos métricas del funcionamiento del programa cuando aumenta el número de procesos. De todas maneras se puede mencionar que la eficiencia se mantiene relativamente constante al incrementar el tamaño del problema.

Balance de cargas:

N	Híbrido (dos máquinas, con 4 threads cada una)
512	B=0.953039
1024	B=0.97848
2048	B=0.989182

En este caso hay 2 niveles de balance de cargas: primero a nivel de los procesos MPI y segundo a nivel de threads de OpenMP. Para este ejercicio sólo tenemos en cuenta las cargas de los dos procesos de MPI.

Las cargas de los procesos están altamente balanceadas, ya que son solo dos.

Impacto de las comunicaciones:

N	Híbrido (dos máquinas, con 4 threads cada una)
512	Tcomunicaciones = 0.117101
1024	Tcomunicaciones = 0.568047
2048	Tcomunicaciones = 2.746431