BENEMERITA UNIVERSIDAD AUTONOMA DE PUEBLA

PROYECTO FINAL

SISTEMAS OPERATIVOS II

Profesor: Luis Enrique Colmenares Guillen

Multiplicacion de matrices

AUTORES

PLACIDO VELAZCO CESAR EDUARDO
201214687

VAZQUEZ MARTINEZ JOSUE
201245534

Reporte Multiplicacion de matrices

REPORTE
Benemerita Universidad Autonoma de Puebla
December 1, 2017

Abstract

En este documento exponemos cada uno de los elementos que hicieron posible la realizacion de el proyecto final de la materia de sistemas operativos II.Ejemplificamos graficamente cada uno de los posibles valores para las matrices sugueridos por el profesor de la materia.

Keywords: OpenMPI ,Paralelo ,Matriz,C

Descripcion de hardware

Alienware 14	
Procesador	Intel Core i7-4700MQ, 4 núcleos (8 hilos) a 2.40 GHz.
Memoria	8 GB
Disco duro	1 TB SATA
Sistema operativo	Ubuntu 16.04

Implementación algoritmo secuencial

1

```
//Creando matrices
int
  A[n][n],
  B[n][n],
  C[n][n];

void fill_matrix(int m[n][n])
{
  int i, j;
  for (i=0; i<n; i++){
    for (j=0; j<n; j++){
      m[i][j] = rand()%(2000 + 1 - 1000)+1000; //Llenamos con
          valores pseudoaleatorios entre 1000 y 2000
    }
  }
}

int main(int argc, char *argv[])
{
  int l,P;
  double average = 0.0;
  MPI_Status status;
  MPI_Init (&argc, &argv);
  for(l = 0; l < 100; l++){ //Iteramos 100 veces

    clock_t start = clock();
    int myrank, from, to, i, j, k;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &P);


    if (n%P!=0) {
      if (myrank==0)
        printf("El tamano de la matriz no es divisible entre el
            numero de procesadores\n");
      MPI_Finalize();
```

[1]Los valores que se corren en el algoritmo son de tipo real, para la prueba con valores de tipo flotante solo basta definir el tipo de valores dentro de el código como float.

```
    exit(-1);
  }

  from = myrank * n/P;
  to = (myrank+1) * n/P;


  if (myrank==0) {
    fill_matrix(A);
    fill_matrix(B);
  }

  MPI_Bcast (B, n*n, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Scatter (A, n*n/P, MPI_INT, A[from], n*n/P, MPI_INT, 0,
     MPI_COMM_WORLD);

  //printf("Resolviendo particion %d (desde el elemento %d
     hasta el %d)\n", myrank, from, to-1);
  for (i=from; i<to; i++){
    for (j=0; j<n; j++) {
      for (k=0; k<n; k++){
        C[i][j] += A[i][k]*B[k][j];
      }
    }
  }
  MPI_Gather (C[from], n*n/P, MPI_INT, C, n*n/P, MPI_INT, 0,
     MPI_COMM_WORLD);
```

Implementación algoritmo de Cannon

```
//Creando matrices
int
  A[n][n],
  B[n][n],
  C[n][n];

void fill_matrix(int m[n][n])
{
  int i, j;
  for (i=0; i<n; i++){
    for (j=0; j<n; j++){
      m[i][j] = rand()%(2000 + 1 - 1000)+1000; //Llenamos con
          valores pseudoaleatorios entre 1000 y 2000
    }
  }
}

int main(int argc, char *argv[])
{
char *fn = argc > 1 ? argv[1] : "gnuplot.dat";

    int pid, statusz;
    float f[MAXL] = {0.0};
    char fnbase[MAXC] = "", fnplt[MAXC] = "";
    size_t iz;
    FILE *fp = NULL;



  int l,P;
  double average = 0.0;
  MPI_Status status;
  MPI_Init (&argc, &argv);
  for(l = 0; l < 100; l++){ //Iteramos 100 veces

    clock_t start = clock();
    int myrank, from, to, i, j, k;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &P);
```

```
if (n%P!=0) {
  if (myrank==0)
    printf("El tamano de la matriz no es divisible entre el
        numero de procesadores\n");
  MPI_Finalize();
  exit(-1);
}

from = myrank * n/P;
to = (myrank+1) * n/P;


if (myrank==0) {
  fill_matrix(A);
  fill_matrix(B);
}

MPI_Bcast (B, n*n, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (A, n*n/P, MPI_INT, A[from], n*n/P, MPI_INT, 0,
   MPI_COMM_WORLD);


double kk;
int vecino_derecho, vecino_izquierdo, vecino_arriba,
    vecino_abajo;

int PP = P;
int x;
double np = P;
kk = sqrt(np);
k = (int)kk;
if (myrank < k) // below neighbour set
{
    vecino_izquierdo = (myrank + k - 1) % k;
    vecino_derecho = (myrank + k + 1) % k;
    vecino_arriba = ((k - 1)*k) + myrank;
}
if (myrank == k)
{
    vecino_izquierdo = ((myrank + k - 1) % k) + k;
    vecino_derecho = ((myrank + k + 1) % k) + k;
    vecino_arriba = myrank - k;
}
```

```
if (myrank > k)
{
    x = myrank / k;
    vecino_izquierdo = ((myrank + k - 1) % k) + x * k;
    vecino_derecho = ((myrank + k + 1) % k) + x * k;
    vecino_arriba = myrank - k;
}
if (myrank == 0 || (myrank / k) < (k - 1))
{
    vecino_abajo = myrank + k;
}
if ((myrank / k) == (k - 1))
{
    vecino_abajo = myrank - ((k - 1)*k);
}
x = 0;

for(int kk = 0; kk < PP; kk++) //algorithm
{
    for (i = 0; i < n / PP; i++)
    {
        for (j = 0; j < n / PP; j++)
        {
            for (k = 0; k < n / PP; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
  }

MPI_Gather (C[from], n*n/P, MPI_INT, C, n*n/P, MPI_INT, 0,
   MPI_COMM_WORLD);
```

Marco comparativo

Dimensiones	Cores	Tiempo		Memoria	
		Secuencial	Cannon	Secuencial	Cannon
16X16	1	0.000060	0.000056	7744 kb	7654 kb
32X32	1	0.000270	0.000359	7724 kb	
64X64	1	0.002067	0.002784	9856 kb	7643 kb
128X128	1	0.014675	0.021790	10172 kb	9235 kb
256X256	1	0.137754	0.170546	8664 kb	7264 kb
512X512	1	1.395750	1.513075	10980 kb	9236 kb
1024X1024	1	15.256436	11.147821	12972 kb	10346 kb
2048X2048	1	125.764687	93.342564	23954682 kb	2295343 kb

Table 1: Comparacion de algoritmos con valores de tipo entero con 1 core.

Dimensiones	Cores	Tiempo		Memoria	
		Secuencial	Cannon	Secuencial	Cannon
16X16	2	0.000044	0.000046	8572 kb	7457 kb
32X32	2	0.000619	0.000148	8576 kb	8554 kb
64X64	2	0.001869	0.000959	9024 kb	8572 kb
128X128	2	0.012188	0.000687	8764 kb	9032 kb
256X256	2	0.154470	0.045504	9680 kb	9420 kb
512X512	2	2.468797	394753	11780 kb	10586 kb
1024X1024	2	4.542345	2.622002	18456 kb	43632 kb
2048X2048	2	135.349123	72.707819	3253465 kb	58088 kb

Table 2: Comparacion de algoritmos con valores de tipo entero con 2 cores.

Dimensiones	Cores	Tiempo		Memoria	
		Secuencial	Cannon	Secuencial	Cannon
16X16	4	0.000039	0.000046	10624 kb	10556 kb
32X32	4	0.000182	0.000148	8536 kb	9112 kb
64X64	4	0.001075	0.000959	10992 kb	8708 kb
128X128	4	0.006956	0.000687	8872 kb	9036 kb
256X256	4	0.057274	0.045504	9149 kb	9304 kb
512X512	4	0.442991	3.94753	10460 kb	10500 kb
1024X1024	4	12.348743	2.622002	324574 kb	6543467 kb
2048X2048	4	89.982345	74.458734	3457528 kb	98745834 kb

Table 3: Comparacion de algoritmos con valores de tipo entero con 4 cores.

Dimensiones	Cores	Tiempo		Memoria	
		Secuencial	Cannon	Secuencial	Cannon
16X16	8	0.000044	0.000037	10816 kb	34567 kb
32X32	8	0.000202	0.000135	8796 kb	42456 kb
64X64	8	0.002413	0.00575	9464 kb	34586 kb
128X128	8	0.010143	0.00277	9284 kb	35678 kb
256X256	8	0.070725	0.020139	9804 kb	3356 kb
512X512	8	0.742964	0.090703	10556 kb	35700 kb
1024X1024	8	5.248933	0.477512	14188 kb	35700 kb
2048X2048	8	16.458642	4.119763	223456 kb	28796 kb

Table 4: Comparacion de algoritmos con valores de tipo entero con 8 cores.

Las matrices utilizadas para cada una de las comparaciones fueron las mismas, con valores generados aleatoriamente en un rango de entre 1000-2000 y 1000.000-2000.000 para las tablas de valores enteros y flotantes respectivamente.
En ambas tablas observamos la comparacion entre tiempo y uso de memoria, siendo evidente que los tiempos son muchisimo mas costosos para el algoritmo secuencial.

Conclusion

El trabajo realizado y presentado de manera comparativa por medio des tablas y la parte de implementacion de cada uno de los algoritmos propuestos por el profesor para la asignatura de sistemas operativos II,nos permitió trabajar con dos algoritmos, todos fueron corridos en una sola maquina portátil, que por sus sobresalientes especificaciones no se presento ningún problema relevante. A medida que se probaban el algoritmo secuencial pudimos ver que en matrices pequeñas el problema no parecía complicado, pero a medida que las dimensiones de las matrices crecían, la presentación de resultados era aun mas tardía. Posteriormente pasamos a la implementación de el algoritmo Cannon, y tuvimos algunos problemas al tratar de correr las primeras pruebas, nos dimos cuenta que algunas versiones permitían el uso de ciertos comandos, eso retraso un poco el avance en la comparación de resultados. La version inicial de Openmpi era la versión 1.02 , posteriormente se instalo la versión 2.0.4 hasta finalmente instalar la ultima que es la 3.0. Como lenguaje de programación utilizamos C, por sus potentes características y su amplio soporte con la librería. De manera general y observando las pruebas en los tiempos que arrojaron las pruebas con cada dimensión de matriz dada, podemos darnos cuenta de la importancia que tiene el paralelismo cuando se trata de problemas donde los valores sean bastantes grandes o sean demasiados.

Bibliografia

@miscASH:2013:Online, author = Varios Autores, title = Archivo Situacionista Hispano, year = 1999, howpublished = urlhttp://sindominio.net/ash, note = Accedido 01-10-2013