Universidad De Granada

Grado ingeniería informática (2017 – 2018)

Programación Paralela

Implementación en memoria compartida y en memoria distribuida de un algoritmo paralelo de datos

Trabajo realizado por Antonio Miguel Morillo Chica.

---

1. Implementación del Algoritmo de Floyd.

1.2. Algoritmo de Floyd Paralelo 1D.

Asumimos que el número de vértices N es múltiplo del número de tareas P. En esta versión, cada tarea procesa un bloque contiguo de filas de la matriz I por lo que cada tarea procesa N/P filas de I. Se podrán utilizar hasta N tareas como máximo. Cada tarea es responsable de una o más filas adyacentes de I y ejecutará el siguiente algoritmo:

```
procedure floyd paralelo 1
begin
I_{i,j} = A
   for k := 0 to N-1
      for i := local_i_start to local_i_start
         for j := 0 to N-1
            I^{k+1}_{i,j} = min{I^k_{i,j}, I^k_{i,k} + I^k_{k,j}}
end;
```

1.2.1. Implementación OpenMP

Basta con repartir equitativamente las iteraciones del segundo bucle del algoritmo secuencial e ntre las hebras. No obstante, se recomienda lo siguiente:

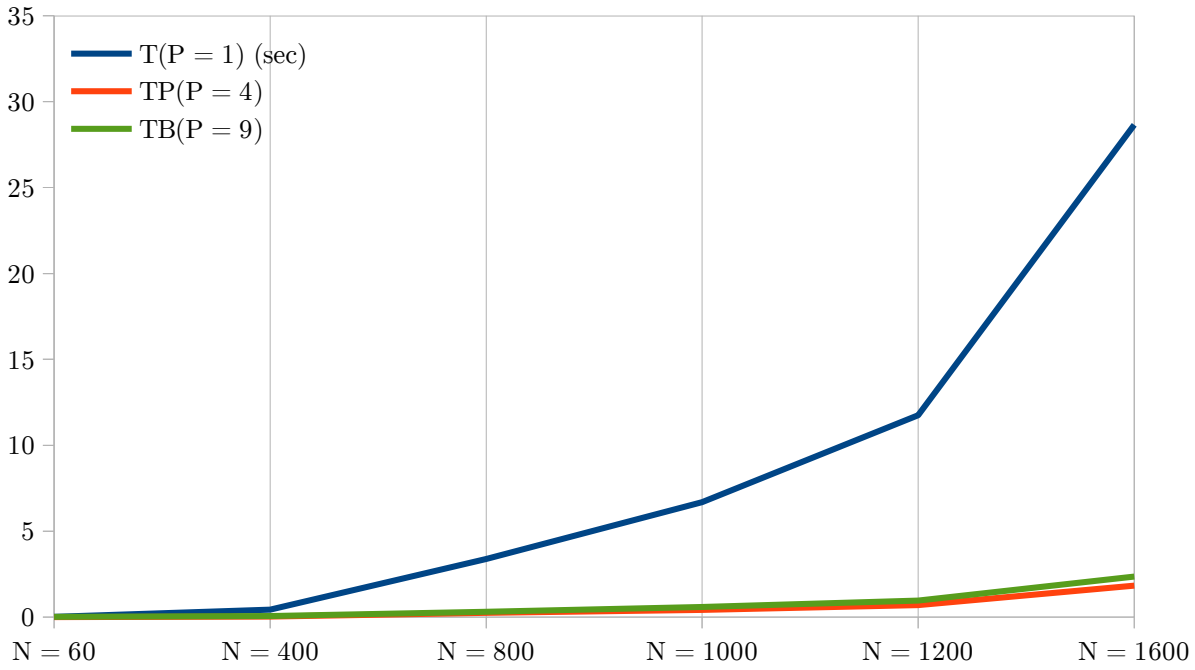
- *Para evitar degradación de eficiencia al acceder a datos compartidos y en el uso de las cachés se recomienda que cada hebra privatice la fila k-ésima, es decir, que copie en su espacio privado dicha fila al comienzo de la k-ésima iteración.*
- *Se aconseja no lanzar una región paralela para cada valor de k sino crear una única región paralela antes del bucle principal y usar una directiva for para cada nuevo valor de k. Hay que tener en cuenta que después de cada #pragma omp for hay un barrier implícito que nos asegura la coherencia de los datos entre una iteración y la siguiente.*

Según lo pedido en la implementación para floyd1D con los datos que nos ha dado es:

```
#pragma omp parallel private(k, i, j, ik, ij, kj, filK)
{
  for (k = 0; k < N; k++) {
    #pragma omp single copyprivate(filK)
    {
      for (i = 0; i < N; i++) {
        filK[i] = M[k * N + i];
      }
    }
   // inicio de la región paralela, reparto estático por bloques
    #pragma omp for schedule(static, chunk)
    for (i = 0; i < N; i++) {
      ik = i * N + k;
      for (j = 0; j < N; j++) {
        if (i != j && i != k && j != k) {
          kj = k * N + j;
          ij = i * N + j;
          M[ij] = min(M[ik] + filK[j], M[ij]);
        }
      }
    }
  }
}
```

1.2.2. Resultados obtenidos.

	T(P = 1) (sec)	T_P(P = 4)	S(P = 4)	T_B(P = 9)	S(P = 9)
N = 60	0,004596	0,0013327	1,21	0,00480246	3,45
N = 400	0,433712	0,0346291	12,52	0,0598331	12,52
N = 800	3,38597	0,246441	13,74	0,301346	13,74
N = 1000	6,68619	0,415394	16,1	0,583338	16,1
N = 1200	11,7534	0,686542	17,12	0,953466	12,33
N = 1600	28,6606	1,82523	15,7	2,35987	12,14



1.3. Algoritmo de Floyd Paralelo 2D.

Por simplicidad, se asume que el número de vértices N es múltiplo de la raíz del número de tareas P. Esta versión del algoritmo de Floyd utiliza un reparto por bloques bidimensionales de la matriz I entre las tareas, pudiendo utilizar hasta N2 procesos. Suponemos que las tareas se organizan lógicamente como una malla cuadrada con $\sqrt{P}$ tareas en cada fila y $\sqrt{P}$ tareas en cada columna. Cada tarea trabaja con un bloque de $N/\sqrt{P}$ subfilas alineadas (cubren las mismas columnas contiguas) con $N/\sqrt{P}$ elementos cada uno y ejecuta el siguiente algoritmo:

```
procedure floyd paralelo 2
begin
Ii,j = A
   for k := 0 to N-1
      for i := local_i_start to local_i_end
         for j := local_j_start to local_j_end
            I k+1 i,j = min{I k i,j , Ik i,k + I k k,j}
end;
```

En cada paso, además de los datos locales, cada tarea necesita $N/\sqrt{P}$ valores de dos procesos localizados en la misma fila y columna respectivamente. Cada una

de las P tareas procesa una submatriz de I de tamaño $N/\sqrt{P} \times N/\sqrt{P}$ (por simplicidad, supondremos que $\sqrt{P}$ divide a N).

1.3.1. Implementación en OpenMP

En este caso no se puede usar la directiva de OpenMP para paralelización de bucles ya que son dos los bucles que se trocean. Por tanto, cada hebra tendrá que identificar explícitamente la parte de la matriz que debe actualizar y sería conveniente lo siguiente:

- *Para mejorar la eficiencia se recomienda que cada hebra copie en su espacio privado antes del bucle principal la parte de la matriz con la que trabajará y que, al final del cómputo, todas las hebras cooperen para escribir sus resultados locales en la matriz N×N. También cada hebra debería privatizar la subfila y la subcolumna que necesita al comienzo de la k-ésima iteración.*
- *Igualmente se aconseja crear una única región paralela. No obstante, debemos tener en cuenta que al final de cada iteración del bucle principal no hay un barrier implícito por lo que debemos hacerlo emplícitamente para asegurar la coherencia de los datos entre iteraciones.*

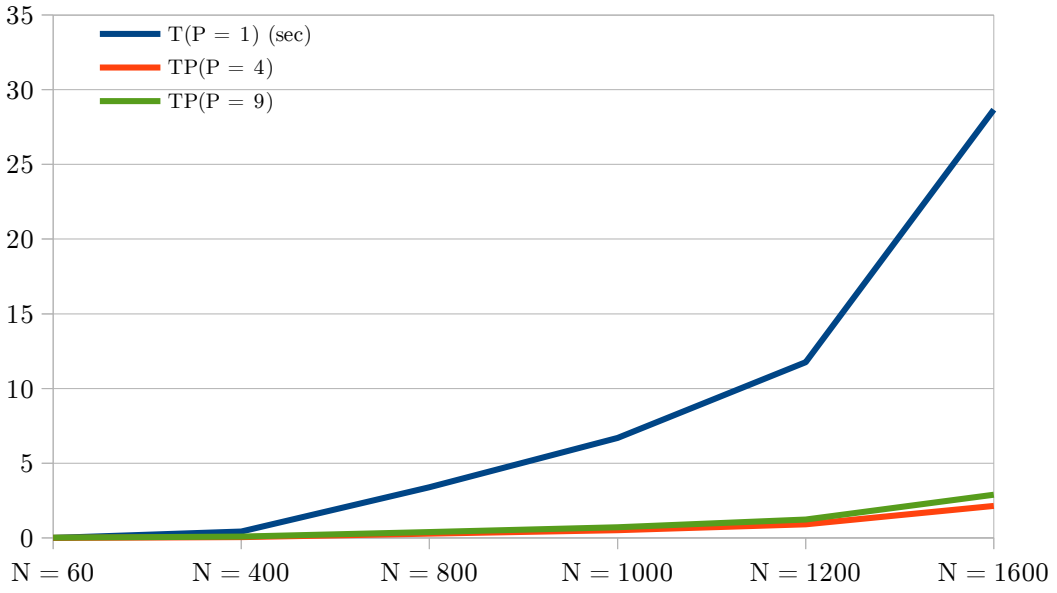
Para esta sección el código implementado según las indicaciones ha sido el siguiente:

```
#pragma omp parallel shared(M,tamBloque,columnak,filak)
private(k,i,j,vikj,iLocalInicio,iLocalFinal,jLocalInicio,jLocalFinal)
{
  int tidDSqrt = omp_get_thread_num()/sqrtP;
  int tidMSqrt = omp_get_thread_num()%sqrtP;

  iLocalInicio = tidDSqrt * tamBloque;
  iLocalFinal = (tidDSqrt+1) * tamBloque;

  jLocalInicio = tidMSqrt * tamBloque;
  jLocalFinal = (tidMSqrt+1) * tamBloque;

  for(k = 0; k<N; k++){
    int kn = k * N;



  #pragma omp barrier
  if (k >= iLocalInicio && k < iLocalFinal)
    for(i = 0; i<N; i++)
      filak[i] = M[ kn + i];
```

```
  if (k >= jLocalInicio && k < jLocalFinal)
    for(i = 0; i<N; i++)
      columnak[i] = M[i * N + k];
  #pragma omp barrier

  for(i = iLocalInicio; i<iLocalFinal; i++){
    int in = i * N;
    for(j = jLocalInicio; j<jLocalFinal; j++){
      if (i != j && i != k && j != k){
        int ij = in + j;
        vikj = columnak[i] + filak[j];
        vikj = min(vikj, M[ij]);
        M[ij] = vikj;
      }
    }
  }
}
```

1.3.1.2 Resultados OpenMP

	T(P = 1) (sec)	T_P(P = 4)	S(P = 4)	T_P(P = 9)	S(P = 9)
N = 60	0,004596	0,00350812	1,21	0,00707126	0,65
N = 400	0,433712	0,0498653	4,8	0,0957777	4,53
N = 800	3,38597	0,284612	11,9	0,391039	8,66
N = 1000	6,68619	0,515643	12,97	0,708525	9,44
N = 1200	11,7534	0,913565	12,87	1,228	9,57
N = 1600	28,6606	2,13034	13,45	2,9	9,9



1.3.2.1 Implementación en MPI.

Teniendo en cuenta las dependencias de datos entre procesos, los requerimientos de comunicación en la etapa k requieren de dos operaciones de broadcast:

- *Desde el proceso en cada fila (de la malla de procesos) que contiene parte de la columna k al resto de procesos en dicha fila.*
- *Desde el proceso en cada columna (de la malla de procesos) que contiene parte de la fila k al resto de procesos en dicha columna.*

En cada uno de los N pasos, N/raiz(P) valores deben difundirse a los raiz(P) procesos en cada fila y en cada columna de la malla de procesos. Nótese que cada proceso debe servir de origen (root) para al menos un broadcast a cada proceso en la misma fila y a cada proceso en la misma columna del malla lógica de procesos bidimensional.

```
for(k = 0; k<nverts; k++)
{
    // idHorizontal y idVertical del comunicador horizontal y vertical
    idProcesoBloqueK = k / tamBloque;
    indicePartidaFilaK = k % tamBloque;

    if (k >= iLocalInicio && k < iLocalFinal)
        copy(subMatriz[indicePartidaFilaK],
             subMatriz[indicePartidaFilaK] + tamBloque, filak);

    if (k >= jLocalInicio && k < jLocalFinal)
        for (i = 0; i < tamBloque; i++)
            columnak[i] = subMatriz[i][indicePartidaFilaK];

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(filak, tamBloque, MPI_INT, idProcesoBloqueK, commVertical);
    MPI_Bcast(columnak, tamBloque, MPI_INT,
              idProcesoBloqueK, commHorizontal);

    for(i = 0; i<tamBloque; i++)
    {
        iGlobal = iLocalInicio + i;
        for(j = 0; j<tamBloque; j++)
        {
            jGlobal = jLocalInicio + j;
            // no iterar sobre la diagonal (celdas a 0)
            if (iGlobal != jGlobal && iGlobal != k && jGlobal != k)
            {
                vikj = columnak[i] + filak[j];
```

```
                    vikj = min(vikj, subMatriz[i][j]);
                    subMatriz[i][j] = vikj;
                }
            }
        }
    }

    // Paramos el cronometro
    t = MPI_Wtime()-t;

    MPI_Barrier(MPI_COMM_WORLD);
   MPI_Gather( subMatriz, tamBloque * tamBloque, MPI_INT, bufferSalida,
               sizeof(int) * tamBloque * tamBloque, MPI_PACKED, 0,
               MPI_COMM_WORLD );
[...]
```

1.3.2.2. Resultados MPI

	T(P = 1) (sec)	T_P(P = 4)	S(P = 4)
N = 60	0,00524993	0,00107198	1,21
N = 400	0,380349	0,14787	4,8
N = 800	2,97029	1,02512	2,9
N = 1000	5,86511	1,98361	2,96
N = 1200	10,3757	3,39659	3,05