Implementación Paralela de la Iteración de Jacobi

César González Segura
cegonse@posgrado.upv.es
Conceptos y Métodos de la Computación Paralela
Máster Universitario en Computación Paralela y Distribuida

Resumen Este trabajo trata la paralelización del algoritmo de Jacobi, utilizando memoria compartida y distribuida. Los algoritmos se han implementado utilizando OpenMP y MPI y se muestran los resultados experimentales obtenidos.

Palabras clave sistemas de ecuaciones, Jacobi, métodos iterativos, OpenMP, MPI.

I. INTRODUCCIÓN

Los sistemas de ecuaciones lineales son un método utilizado muy frecuentemente para expresar problemas en física, ingeniería, ciencias sociales, etc. La complejidad que alcanzan estos problemas requiere algoritmos que sean capaces de obtener su solución computacionalmente en un tiempo razonable.

El algoritmo de Jacobi ofrece una solución iterativa a la resolución de sistemas de ecuaciones lineales, obteniendo una aproximación a la solución real del sistema. La calidad de esta solución puede ajustarse, obteniendo mejores resultados a costa de un mayor tiempo de cálculo.

Un sistema de ecuaciones con *n* ecuaciones y *n* incógnitas puede representarse en forma matricial como:

$$\mathrm{Ax} = \mathrm{b} \equiv \equiv \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Donde *A* representa la matriz de coeficientes del sistema, *b* el vector de términos independientes y *x* el vector de incógnitas. Tomando la matriz inversa de *A*, puede obtenerse la solución del sistema:

$$\mathrm{Ax} = \mathrm{b} \equiv \mathrm{x} = \mathrm{A}^{-1}\mathrm{b}$$

Para el desarrollo de la iteración de Jacobi, se reordenan las ecuaciones separando la matriz *A* en la suma de su parte triangular superior e inferior en la matriz *R* y en su matriz diagonal, *D*:

$$\mathrm{A} = \mathrm{L} + \mathrm{D} + \mathrm{U}$$
$$\mathrm{R} = \mathrm{L} + \mathrm{U}$$

Con estos parámetros, puede reescribirse la ecuación de forma iterativa obteniendo, siendo *k* la solución en la iteración actual:

$$\mathrm{x}^{(\mathrm{k}+1)} = \mathrm{D}^{-1}\left[\mathrm{b} - \mathrm{Rx}^{(\mathrm{k})}\right]$$

Para obtener la solución al sistema, se parte de una aproximación inicial a la solución ($x^{(0)}$) y se substituye en cada iteración hasta obtener una solución tan próxima como se desee. Este parámetro de convergencia se puede estimar como:

$$\left\|\mathrm{Ax}^{(\mathrm{k}+1)} - \mathrm{b}\right\| < \varepsilon$$

Siendo ε un valor de error mínimo definido. La convergencia de la solución está asegurada siempre que su radio espectral sea menor que uno:

$$\rho(\mathrm{D}^{-1} \cdot \mathrm{R}) < 1$$

En la implementación de los algoritmos, se fuerza a que la matriz de coeficientes del sistema sea de diagonal dominante positiva, lo cual hará que la condición anterior se cumpla.

En la siguiente sección se expone el diseño del algoritmo en forma secuencial y en forma paralela, utilizando memoria compartida y memoria distribuida.

II. DISEÑO ALGORÍTMICO

El diseño de los algoritmos se ha separado en el diseño del algoritmo secuencial, en memoria compartida y en memoria distribuida.

Para el diseño en memoria compartida se ha utilizado el modelo teórico *P-RAM* (Parallel Random Access Machine) utilizando un modelo de acceso a datos *CRCW* (lectura concurrente y escritura concurrente).

Para el diseño del algoritmo en memoria distribuida, se ha utilizado el modelo teórico de paso de mensajes *DCM*, donde la transmisión de mensajes de un nodo a otro tiene un coste de comunicación $n\tau + \beta$.

El algoritmo en forma secuencial puede expresarse de la siguiente manera:

Función Jacobi

Entradas: Doble A[n,n], Doble b[n], Doble x(0)[n], Doble ε, Doble n
Salidas: Doble xs[n]

Si no isdom(*A*, *n*) **hacer**
  Salir
Fin Si

Para *i* = 0 **hasta** *n* – 1 **hacer**
  Para *j* = 0 **hasta** *n* – 1 **hacer**
    Si *i* = *j* **hacer**
      Dinv (*i*) ← 1 / *A*(*i*, *j*)
    Si no
      R(*i*, *j*) ←*A*(*i*, *j*)
    Fin Si
  Fin Para
Fin Para

Para *i* = 0 **hasta** *n* – 1 **hacer**
  $C(i) \leftarrow Dinv(i) \cdot b(i)$
Fin Para

$conv \leftarrow \varepsilon + 1$
$x^{(k)} \leftarrow x^{(0)}$

Mientras $conv > \varepsilon$ **hacer**
  $x^{(k+1)} \leftarrow R \cdot x^{(k)}$

  Para *i* = 0 **hasta** *n* – 1 **hacer**
    $x^{(k+1)}(i) \leftarrow -Dinv(i) \cdot x^{(k+1)}(i)$
    $x^{(k+1)}(i) \leftarrow x^{(k+1)}(i) + C(i)$
  Fin Para

  $x^{(k)} \leftarrow x^{(k+1)}$
  $conv \leftarrow ||A \cdot x^{(k+1)} - b||$
Fin Mientras

$xs \leftarrow x^{(k+1)}$

El primer bloque del algoritmo comprueba que la matriz de coeficientes del sistema sea de diagonal dominante mediante una función auxiliar[1], abortando la ejecución en caso contrario.

Después, se obtiene la matriz *R* y la inversa de la matriz *D*. Para obtener esta última se aprovecha la propiedad de la inversión de matrices diagonales:

$$D^{-1} = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix}^{-1} = \begin{bmatrix} 1/d_1 & 0 & \cdots & 0 \\ 0 & 1/d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/d_n \end{bmatrix}$$

Se ha almacenado la inversa de la matriz *D* en un vector unidimensional para ahorrar espacio. Para obtener el vector *C*, se aprovecha también que el producto matriz-vector se realiza con una matriz diagonal.

A continuación se inicializa la solución actual con la solución inicial, y se entra en el bucle. El bucle calcula la solución en el instante siguiente, y encuentra el valor de convergencia. Se repetirá hasta que se llegue al valor especificado.

Una vez se sale del bucle, se entrega la solución obtenida. Se ha estudiado el coste del algoritmo, teniendo en cuenta únicamente las operaciones aritméticas en coma flotante, siendo:

$$O(\text{Jacobi}) = \text{iter} * (4n^2 + 5n + 1) + 3n^2 + 2n + 1$$

En base al algoritmo secuencial se diseña el algoritmo en memoria compartida. Para paralelizar el algoritmo se ha dividido el problema en bloques fila.

Cada uno de los procesadores estará encargado de obtener sólo un bloque de la solución total. De esta manera, se consigue balancear la carga entre los procesadores disponibles.

Siendo *p* el número total de procesadores y *pr* el índice de procesador actual, el algoritmo paralelo en memoria compartida se puede expresar como a continuación.

Función Jacobi

Entradas: Doble A[n,n], Doble b[n], Doble x(0)[n], Doble ε, Doble n, Entero pr, Entero p
Salidas: Doble xs[n]

Si no isdom(A, n, pr) **hacer**
  Salir
Fin Si

$tb \leftarrow n / p$

Para $i = 0$ **hasta** $tb - 1$ **hacer**
  Para $j = 0$ **hasta** $n - 1$ **hacer**
    Si $i = j$ **hacer**
      $Dinv\ (i) \leftarrow 1 / A(i, j)$
    Si no
      $R(i, j) \leftarrow A(i, j)$
    Fin Si
  Fin Para
Fin Para

Para $i = 0$ **hasta** $tb - 1$ **hacer**
  $C(i + tb*pr) \leftarrow Dinv(i + tb*pr) \cdot b(i + tb*pr)$
Fin Para

$conv \leftarrow \varepsilon + 1$
$x^{(k)}(tb*pr{:}tb*(1 + pr)) \leftarrow x^{(0)}(tb*pr{:}tb*(1 + pr))$

Mientras $conv > \varepsilon$ **hacer**
  $x^{(k+1)}(tb*pr{:}tb*(1 + pr)) \leftarrow R \cdot$
    $x^{(k)}(tb*pr{:}tb*(1 + pr))$

  Para $i = 0$ **hasta** $tb - 1$ **hacer**
    $x^{(k+1)}(i + tb*pr) \leftarrow -Dinv(i + tb*pr) \cdot$
      $x^{(k+1)}(i + tb*pr)$
    $x^{(k+1)}(i + tb*pr) \leftarrow x^{(k+1)}(i + tb*pr) +$
      $C(i + tb*pr)$
  Fin Para

  $x^{(k)}(tb*pr{:}tb*(1 + pr)) \leftarrow x^{(k+1)}(tb*pr{:}tb*(1 + pr))$
  $conv \leftarrow ||A \cdot x^{(k+1)}(tb*pr{:}tb*(1 + pr)) -$
    $b(tb*pr{:}tb*(1 + pr))||$
Fin Mientras

$xs \leftarrow x^{(k+1)}$

Dividiendo el problema en bloques fila se acelera el algoritmo en las secciones que pueden calcularse de forma paralela.

Al igual que con el algoritmo secuencial, se evalúan los costes para conocer el rendimiento, los cuales son:

$$O(\text{Jacobi}) = \text{iter} * \left(4n^2 + \frac{5n}{p} + 1\right) + \frac{3n^2}{p} + \frac{2n}{p} + p + 3$$

El algoritmo en memoria compartida comparte la misma filosofía que en memoria compartida, sólo que en este caso es necesario transmitir los datos a los diferentes nodos.

Este algoritmo asume que van a existir al menos dos nodos de proceso. El nodo cero enviará los datos de entrada al resto de nodos: la matriz A se enviará completa, y de los vectores b y $x^{(0)}$ sólo se enviará el bloque que corresponda a cada nodo.

Una vez obtenidos todos los datos, la iteración empieza, con la diferencia entre este algoritmo y el algoritmo en memoria compartida dada por la existencia de un *lock-step*.

El programa no pasará a la siguiente iteración hasta que todos los nodos hayan calculado el bloque de la solución que les corresponda, ya que el nodo cero es el responsable de calcular el valor de convergencia.

El algoritmo en memoria distribuida se describe a continuación.

Función Jacobi

Entradas: Doble A[n,n], Doble b[n], Doble x(0)[n], Doble ε, Doble n, Entero pr, Entero p
Salidas: Doble xs[n]

$tb \leftarrow n / p$

Si no isdom(A, n) **hacer**
  Salir
Fin Si

Si $pr = 0$ **hacer**
  Para $i = 0$ **hasta** $i = p - 1$ **hacer**
    Enviar(A, i)
    Enviar($b(tb*i{:}tb*(i+1))$, i)
    Enviar($x^{(0)}(tb*i{:}tb*(i+1))$, i)
  Fin Para
Si no hacer
  Recibe(A)
  Recibe(b)
  Recibe($x^{(0)}$)
Fin Si

Para $i = 0$ **hasta** $tb - 1$ **hacer**
  Para $j = 0$ **hasta** $n - 1$ **hacer**
    Si $i = j$ **hacer**

```
        Dinv (i + tb*pr) ← 1 / A(i + tb*pr, j)
      Fin Si
    Fin Para
Fin Para

Para i = 0 hasta n – 1 hacer
    Para j = 0 hasta n – 1 hacer
      R(i, j) ←A(i, j)
    Fin Para
Fin Para

Para i = 0 hasta tb – 1 hacer
    C(i + tb*pr)  ← Dinv(i + tb*pr) ·b(i + tb*pr)
Fin Para

conv  ←ε + 1
continuar  ← si
x(k)(tb*pr:tb*(1 + pr))  ← x(0)(tb*pr:tb*(1 + pr))

Mientras continuar = si hacer
    Si pr = 0 hacer
      Para i = 1 hasta i = p – 1 hacer
        Recibe(x(k)(tb*i:tb*(i+1)))
      Fin Para

      conv  ← ||A · x(k) – b||

      Si conv < ε hacer
        continuar  ← no
      Fin Si

      Para i = 0 hasta i = p – 1 hacer
        Envia(continuar, i)
      Fin Si
Si no
    x(k+1)(tb*pr:tb*(1 + pr)) ← R ·
      x(k)(tb*pr:tb*(1 + pr))

    Para i = 0 hasta tb – 1 hacer
      x(k+1)(i + tb*pr) ← −Dinv(i + tb*pr) ·
        x(k+1)(i + tb*pr)
      x(k+1)(i + tb*pr) ← C(i + tb*pr) +
        x(k+1)(i + tb*pr)
    Fin Para

    x(k)(tb*pr:tb*(1 + pr)) ← x(k+1)(tb*pr:
      tb*(1 + pr))

    Para i = 0 hasta i = p – 1 hacer
      Si i ≠ j hacer
        Envia(x(k)(tb*i:tb*(i+1)), i)
      Si no
        Recibe(x(k)(tb*i:tb*(i+1)), i)
      Fin Si
    Fin Para

    Recibe(continuar, 0)
    Fin Si
Fin Mientras

xs  ← x(k+1)
```

De la misma manera que con los otros dos algoritmos, se obtiene el coste del algoritmo. Debido a las comunicaciones habrá dos costes, el coste aritmético y el coste de comunicación, los cuales son:

$$O(\text{Jacobi}_{\text{comm}}) = \text{iter} * [\tau(n+1) + \beta(p+1)] + \tau\left(n^2 + \frac{2n}{p}\right) + 3\beta$$

$$O(\text{Jacobi}_{\text{arit}})\text{iter} * \left[n^2\left(3 + \frac{2}{p}\right) + n\left(1 + \frac{1}{p}\right) + 1\right] + n^2\left(2 + \frac{1}{p}\right) + n\left(1 + \frac{1}{p}\right) + 2$$

III. IMPLEMENTACIÓN

Los tres algoritmos se han implementado en el lenguaje C para comprobar su validez.

Se han utilizado las librerías OpenMP y MPI para implementar los algoritmos en memoria compartida y memoria distribuida respectivamente.

Las operaciones de álgebra matricial se han implementado manualmente para evitar que las estadísticas puedan verse afectadas por los cálculos, pero utilizando una sintaxis similar a las librerías BLAS para que puedan ser sustituidas.

El código fuente puede consultarse en el paquete adjunto con este documento o en GitHub. En el paquete se incluyen los archivos *Makefile* para la compilación de los distintos programas y scripts para realizar pruebas de tiempos.

IV. PRUEBAS EXPERIMENTALES

Se han probado los algoritmos secuenciales, en memoria compartida y en memoria distribuida.

La batería de pruebas ha consistido en obtener los tiempos necesarios para hallar la solución del sistema con tamaños de matriz desde 128 hasta 5000.

Las pruebas se han realizado 3 veces, calculando el tiempo medio de las 3 ejecuciones, en instantes de tiempo en los que la carga del sistema era lo más estable posible.

Las pruebas se han realizado en el sistema *quadcluster* de la *Universitat Politècnica de València*, con las siguientes características:

1. Un nodo frontend con cuatro núcleos *Intel Xeon* a 2.33 GHz y 8 GB de memoria RAM.
2. Cuatro nodos worker con ocho núcleos *Intel Xeon* a 3 GHz y 16 GB de memoria RAM cada uno.

Los valores experimentales obtenidos de las pruebas se pueden consultar en la hoja de cálculo adjunta a este documento.

VII. RESULTADOS

A continuación se muestran los resultados obtenidos en la ejecución del algoritmo en memoria compartida y en memoria distribuida.

Para memoria compartida, con un número de procesadores variando entre dos y ocho se han obtenido los tiempos siguientes, comparados con el tiempo secuencial:

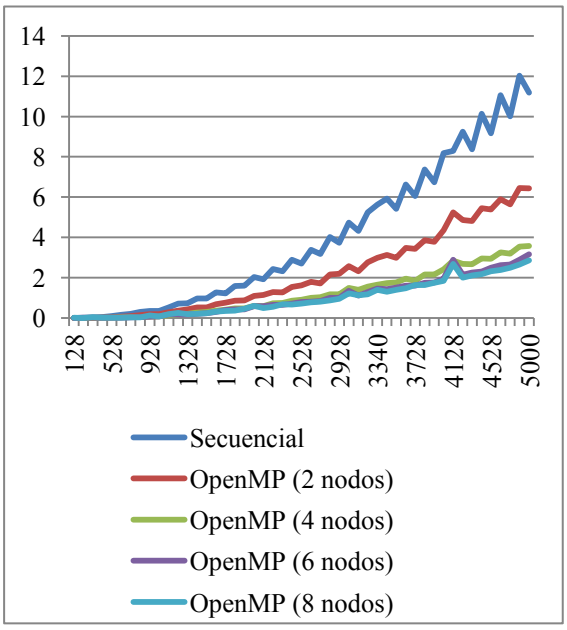
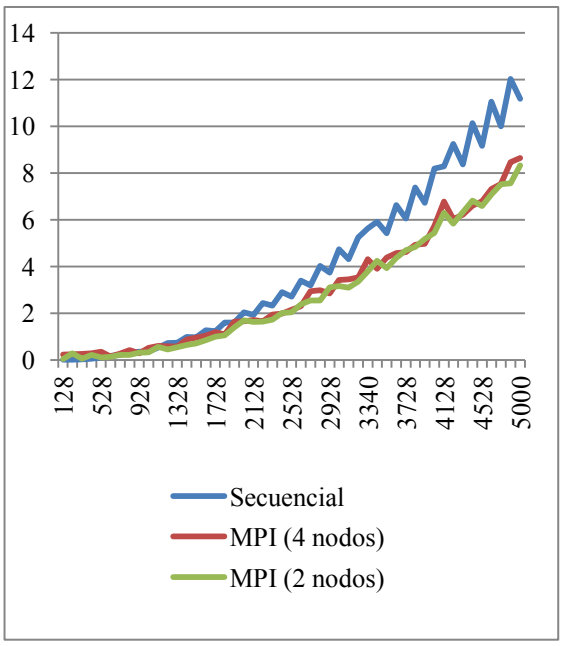


Figura 1: Prestaciones en memoria compartida.

El cálculo en memoria compartida obtiene mejores prestaciones que el cálculo secuencial, encontrándose la solución en aproximadamente 3 segundos en el peor caso de memoria compartida y en 11 en el peor caso secuencial.

Para memoria distribuida, se ha probado el algoritmo utilizando cuatro y dos nodos MPI, y la comparación con el tiempo secuencial es la siguiente:



En memoria distribuida también se ha obtenido una mejora con respecto al tiempo secuencial, tanto con cuatro como con dos nodos. Sin embargo, el tiempo de ambos es muy similar.

A falta de realizar más pruebas, la razón puede estar en la existencia de *lock-step* en el algoritmo, impidiendo la aceleración al escalar horizontalmente.

V. CONCLUSIONES

Se ha demostrado que el algoritmo de Jacobi puede paralelizarse tanto en memoria compartida como en memoria distribuida, obteniendo mejores prestaciones que en el cálculo secuencial.

Una posible mejora al trabajo realizado sería rediseñar el algoritmo en memoria distribuida para minimizar el efecto del *lock-step* y agilizar su ejecución.

VI. REFERENCIAS

[Documentación librería MPI]
http://www.mpich.org/static/docs/v3.1/www3/

[Referencia rápida OpenMP]
http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf