Programación multinúcleo: Inversión de Matrices

Diego Galíndez Barreda (A01370815)
Tecnológico de Monterrey, Campus Estado de México
A01370815@itesm.mx

8 de noviembre de 2015

Resumen

Este documento, escrito en LaTeX, presenta un artículo de investigación sobre la implementación de una inversión de matrices por método de gauss jordan sin el uso de pivotes, para la materia de *Programación multinúcleo* del Instituto Tecnológico de Estudios Superiores de Monterrey, Campus Estado de México.

1. Resumen

El presente trabajo se basa en una implementación de la inversión de una matriz por el método de Gauss Jordan [1], sin usar pivotes.

Se busca demostrar las ventajas de las implementaciones de algoritmos que funcionan en paralelo en los diferentes paradigmas de sincronización usados.

En este caso se demuestran las diferencias con paso de mensajes y memoria compartida.

Para llevar a cabo la comparación de tiempos, se realizó el algoritmo de forma secuencial y a continuación, de forma concurrente. Además, por cada algoritmo se realizaron varias corridas para poder determinar un promedio y calcular el *speedup* del sistema implementado de forma concurrente contra el secuencial.

Los lenguajes y tecnologías usadas para ellos fueron:

- *Web Workers* en JavaScript
- *Streams* paralelos en Java 8
- Procesos en Erlang

Durante el presente artículo se determina que, para el caso particular de este algoritmo, los sistemas de memoria compartida resultan más eficientes que los de paso de mensajes.

Además, con este algoritmo, se puede ver claramente la influencia de la Ley de Amdahl [2], ya que no toda la parte computacionalmente costosa es paralelizable, lo cual limita el aumento en velocidad que se llega a obtener.

Finalmente, se concluyen las ventajas y desventajas ante diferentes algoritmos dentro de los sistemas concurrentes y se destaca la importancia del conocimiento y entendimiento del desarrollo concurrente hoy en día para aprovechar las tecnologías actuales.

2. Introducción

Durante el presente trabajo se llevó a cabo una inversión de matrices por el método de Gauss Jordan sin pivotes. A continuación se explica a detalle el funcionamiento básico del algoritmo así como señalar las partes del mismo que pueden ser implementadas de forma paralela.

En un principio, se aumenta la matriz original con la matriz identidad de su mismo tamaño, de forma que se obtiene una matriz de $N \times 2N$. [1]

Matriz original:

$$\begin{matrix} a_{00} & a_{01} & \dots & a_{0N} \\ a_{10} & a_{11} & \dots & a_{1N} \\ \vdots & & & \\ \vdots & & & \\ \vdots & & & \\ a_{N0} & a_{N1} & \dots & a_{NN} \end{matrix}$$

Matriz extendida:

$$\begin{matrix} a_{00} & a_{01} & \dots & a_{0N} & 1 & 0 & \dots & 0_{0,2N} \\ a_{10} & a_{11} & \dots & a_{1N} & 0 & 1 & \dots & 0_{0,2N} \\ \vdots & & & & & & & \\ \vdots & & & & & & & \\ \vdots & & & & & & & \\ a_{N0} & a_{N1} & \dots & a_{NN} & 0 & 0 & \dots & 1_{0,2N} \end{matrix}$$

En el método de eliminación Gaussiana, se buscan los pivotes por cada renglón, los cuales son indicados por el valor más alto de la columna sobre la que se está iterando. Una vez encontrados los pivotes, se invierte el orden del renglón en el cual se encontró el pivote por el renglón igual a la columna sobre la que se está iterando.

Pivote en la fila 2, revisando la columna 0:

$$\begin{matrix} a_{00} & a_{01} & \dots & a_{0N} \\ a_{10} & a_{11} & \dots & a_{1N} \\ a_{20} & a_{21} & \dots & a_{2N} \\ \vdots & & & \\ \vdots & & & \\ a_{N0} & a_{N1} & \dots & a_{NN} \end{matrix}$$

Por cada cambio que se realice, se debe guardar el cambio que se hizo, ya que estos provocan diferentes transformaciones sobre la matriz. Después de esto se realizan operaciones de renglón sobre la matriz de forma tal que quede en forma triangular superior, esto es, que los valores debajo de la diagonal principal sean todos ceros.

Matriz en forma triangular superior:

$$\begin{matrix} a_{00} & a_{01} & \dots & a_{0N} \\ 0 & a_{11} & \dots & a_{1N} \\ 0 & 0 & \dots & a_{2N} \\ \vdots & & & \\ \vdots & & & \\ 0 & 0 & \dots & a_{NN} \end{matrix}$$

La alternativa a esta opción, es el método de Gauss Jordan, que consiste en hacer que la matriz original termine como la identidad. Esto se consigue haciendo sustitución en reversa sobre la parte superior a la diagonal, de forma que se consigue que todos estos valores sean 0.

Matriz reducida:

$$\begin{matrix} a_{00} & 0 & \dots & 0_{0N} \\ 0 & a_{11} & \dots & 0_{1N} \\ 0 & 0 & \dots & 0_{2N} \\ \vdots & & & \\ \vdots & & & \\ 0 & 0 & \dots & a_{NN} \end{matrix}$$

Finalmente, se debe garantizar que los valores de la diagonal sean 1. Todas las transformaciones realizadas sobre la matriz se llevan a cabo también sobre la identidad. Esto implica que los valores de la identidad son los valores que deberá tener la matriz invertida al finalizar el algoritmo.

Una alternativa a este método, radica en evitar los pivotes. Esto se puede hacer simplemente dividiendo cada renglón entre el valor que debería quedar en su diagonal, `a_{ii}`. Esto provoca que por definición, todos los valores de la diagonal queden en 1 y se simplifica ampliamente el proceso.

Considerando los factores anteriormente expuestos, se pueden observar dos aspectos importantes: las oportunidades de paralelismo y la diferencia entre el uso de pivotes y su ausencia.

Primero que nada, podemos observar que existen varias oportunidades para realizar el algoritmo de forma paralela: la generación de la matriz identidad y su unión a la matriz base, las operaciones de renglón para obtener las escalas y la sustitución en reversa. Sin embargo, todas estas requieren que los valores sean colocados en la matriz y resulta importante garantizar que al realizar este paso no se presenten condiciones de carrera, ya que es posible que dos diferentes hilos intenten escribir el mismo valor.

Por otra parte, es importante destacar que la ausencia de pivotes, aunque agrega más pasos de computación al algoritmo, a la larga evita que se tengan que hacer más operaciones, ya que se tiene garantizado el que todos los valores de la diagonal son la unidad y por lo mismo no requerimos mantener una lista de cambios en los renglones y las transformaciones que estos cambios de lugar de cada renglón provocarían.

El realizar la división de cada renglón entre el valor de su diagonal es $O(N)$, ya que solamente se tiene que recorrer la matriz en una dimensión y solamente se realiza una vez. Por otra parte, la búsqueda de pivotes, asumiendo que se haga de forma binaria, toma $O(N \log N)$, ya que cada búsqueda tomaría $O(\log N)$ y se tendría que hacer una búsqueda por cada renglón. Aunado a esto, se tiene que considerar la memoria extra para almacenar las operaciones y el tiempo que tome hacer el cambio de renglón, lo cual dependería de la implementación particular.

Por lo estipulado anteriormente, se decidió usar la eliminación de GaussJordan sin pivotes, de forma que se pueda aprovechar en la mayor medida posible la paralelización.

Se pudo determinar que las áreas donde se puede aprovechar principalmente la paralelización son: la creación de la matriz de identidad y su unión a la matriz original, las escalas de cada renglón para crear la diagonal y la sustitución en reversa de las escalas obtenidas. Resulta de gran importancia realizarlo de la forma correcta y asegurarse de que no van a existir condiciones de carrera en los sistemas de memoria compartida.

3. Desarrollo

Para realizar las comparaciones de este algoritmo, se utilizaron las siguientes tecnologías:

- *Web Workers* en JavaScript
- *Streams* paralelos en Java 8
- Procesos en Erlang

En cada uno de estos se realizó la implementación secuencial y concurrente del algoritmo. Para poder realizar una comparación entre estos, se llevaron a cabo 5 corridas de cada programa, por cada una de las implementaciones. De estas se obtuvo un promedio y se calculó el *speedup* para cada una de las tres implementaciones. Esto nos permite comprar las ventajas de cada lenguaje independientemente de su velocidad natural.

Se adaptaron los tamaños usados para la matriz para cada lenguaje, ya que en algunas de las implementaciones no era viable usar matrices de cierto tamaño. Aunado a ello, cada una de las implementaciones comienza generando una matriz con números aleatorios que se usará tanto para el caso secuencial como el paralelo. Esto garantiza que sean la misma cantidad de operaciones independientemente de la versión que se esté usando.

Las especificaciones técnicas del equipo utilizado para estos fines son:

- *P*rocesador Intel Core i5-4200U de 1.60GHz con dos núcleos y cuatro hyperthreads.
- *5*.7 GiB de memoria RAM.
- *S*istema operativo Windows 8.1, de 64 bits.
- *N*avegador Mozilla Firefox 40.0.3.
- *C*ompilador Java 1.8.0_51 de Oracle.
- *E*rlang/OTP 18 [erts-7.0] de 64 bits.

Primero que nada, se llevo acabo la implementación del algoritmo en `JavaScript`. En un principio se hizo el algoritmo secuencial sin considerar su paralelización. Este se usó como base para la implementación paralela.

Se crearon dos `WebWorkers` diferentes para realizar la versión paralela. El primero de estos se encargaba de crear la matriz de identidad del tamaño de la matriz generada y agregarla a la matriz original. Dentro del programa principal se obtenía la información que generaba este hilo y se juntaba a la matriz real. Esto era necesario debido a que este es un sistema de paso de mensaje, no de memoria compartida.

A continuación se realiza como tal el algoritmo de GaussJordan. Como se mencionó anteriormente, en este proceso se puede paralelizar el obtener las escalas y la sustitución en reversa de los valores. Esto se realizó dentro de un mismo `WebWorker` que, de acuerdo a la información que recibía en los mensajes, determinaba si correspondía obtener las escalas o realizar la sustitución.

Para esta implementación se usó una matriz de 150×150 y 4 `WebWorkers` en la versión concurrente y solamente 1 `WebWorker` para la versión secuencial. Los resultados de esta implementación se muestran a continuación:

Resultados de la ejecución secuencial:

# Corrida	Tiempo (segundos)
1	4.91
2	4.51
3	5.08
4	4.66
5	4.95

Promedio: 4.82 segundos

Resultados de la ejecución concurrente:

# Corrida	Tiempo (segundos)
1	5.96
2	5.97
3	5.83
4	5.95
5	5.81

Promedio: 5.90 segundos Speedup: 0.81X

Como se puede observar, la implementación paralela resulta más lenta que la secuencial, a pesar de que se buscó aislar la parte más computacionalmente costosa dentro de la parte paralelizada. Esto es debido a que, al ser un modelo de memoria compartida, es necesario colocar la información obtenida del mensaje dentro de la matriz original. Esto resulta en un gran problema, ya que, a pesar de que se aísla el cálculo a los hilos creados, sigue siendo necesario iterar sobre los valores obtenidos. Si a esto se agrega el costo generado por el paso de los mensajes a los `WebWorkers`, termina resultando más costosa la implementación paralela.

Al presentarse un *speedup* negativo menor a 1 podemos determinar que de hecho no existe un *speedup*, se pierde velocidad. Si a esto se agrega la complejidad de la implementación en paralelo en comparación con la secuencial es claro que no es conveniente implementar este algoritmo con `JavaScript`.

La siguiente implementación realizada fue la de `Java`, usando `Streams` para la concurrencia. Se decidió usar esta tecnología en particular, porque simplifica la paralelización de ciclos o funciones recursivas de forma significativa.

De la misma manera que con la implementación de `JavaScript`, se inició con la versión secuencial del algoritmo. En este caso se conservó la versión secuencial para poder realizar las comparaciones, ya que los `Streams` aíslan la paralelización y no se podría usar la misma lógica que en el caso anterior.

Después de realizar la versión secuencial se buscó paralelizarla. Esto resultó muy sencillo, ya que el algoritmo consiste en varios ciclos con iteraciones definidas (`for`) y convertir estos en un `Stream` de `Java` resulta casi trivial.

Como se mencionó en el caso anterior, se paralelizó la creación de la matriz de identidad, su unión a la original, así como la creación de las escalas y la sustitución en reversa de los valores. Sin embargo, contrario a la implementación de memoria compartida, no es necesario actualizar la matriz una vez que los cálculos de cada parte son completados.

En este caso, resulta importante garantizar que los limites de cada división no se empalmen unos con los otros, ya que esto llevaría a condiciones de carrera y, por lo mismo, provocaría resultados no deseados ni esperados.

Para esta implementación se usó una matriz de 1000×1000 tanto para la versión concurrente como para la versión secuencial. Los resultados de esta implementación se muestran a continuación:

Resultados de la ejecución secuencial:

# Corrida	Tiempo (segundos)
1	1.74
2	1.81
3	1.63
4	1.63
5	1.69

Promedio: 1.7 segundos

Resultados de la ejecución concurrente:

# Corrida	Tiempo (segundos)
1	0.73
2	0.78
3	0.69
4	0.77
5	0.78

Promedio: 0.75 segundos Speedup: 2.33X

Inmediatamente se puede observar el gran aumento en la velocidad de ejecución del algoritmo en el caso concurrente. El *speedup* que se obtiene es de $2.33\times$, lo cual implica que el algoritmo concurrente toma un poco menos de la mitad del tiempo en ser ejecutado.

Esta diferencia tan significativa en comparación con la implementación de `WebWorkers` se debe en gran parte al sistema de memoria compartida usado por `Java` contra el de paso de mensajes. En este caso, el costo provocado por la creación de los hilos para la paralelización del programa, no resulta tan fuerte como para ser detrimental para el tiempo de ejecución. Además, al no requerir el envío y copia de la información que se usará en cada hilo, se disminuye considerablemente el costo total del programa.

Es claro que la implementación paralela no tiene una complejidad mucho mayor a la secuencial y el aumento de velocidad es bastante considerable. Esto nos da como conclusión lógica que, en este caso, resulta mucho más conveniente el uso de la implementación paralela.

Para concluir estas pruebas, se realizó la implementación con `Erlang` de forma secuencial y concurrente, en el caso de la forma concurrente se hizo uso del módulo de `plists`, el cual ofrece implementaciones de las funciones predefinidas en las listas de `Erlang` de forma concurrente. Esto facilita de forma considerable la implementación.

La principal motivación para escojer `Erlang`, a pesar de los resultados de la implementación de `JavaScript`, es la comparación entre los `lightweight processes` de `Erlang` y la creación de `WebWorkers` en `JavaScript`. Esto nos permite observar la diferencia en la eficiencia de los procesos de `Erlang` contra la creación de procesos o hilos diferentes.

A continuación se presentan los resultados de la implementación, usando una matriz de 150×150, principalmente para poderlo comparar de forma efectiva con la implementación de `JavaScript`:

Resultados de la ejecución secuencial:

# Corrida	Tiempo (segundos)
1	0.454
2	0.531
3	0.516
4	0.453
5	0.500

Promedio: 0.49 segundos

Resultados de la ejecución concurrente:

# Corrida	Tiempo (segundos)
1	1.343
2	1.125
3	1.281
4	1.203
5	1.297

Promedio: 1.249 segundos Speedup: 0.39X

Como se puede observar, la versión secuencial es considerablemente más eficiente que la concurrente. De hecho en este caso se ve que la versión concurrente es considerablemente más lenta. Esto deja claro que no es conveniente realizar el algoritmo en un sistema que funciona por paso de mensajes.

Sin embargo, se puede observar que, a pesar de que el *speedup* es considerablemente menor que el de cualquiera de las otras implementaciones, ambas versiones son más rápidas que sus equivalentes en `JavaScript`. Esto nos permite determinar también que `Erlang` realmente resulta más eficiente que otras implementaciones que funcionan por paso de mensajes. Esto también resulta de importancia, porque nos permite ver que la máquina virtual es realmente muy eficiente en su funcionamiento.

4. Conclusiones

Al llevar a cabo la implementación de este algoritmo, tanto de forma secuencial como concurrente, pude observar las ventajas prácticas de los sistemas de memoria compartida en comparación con los de paso de mensajes. También pude observar la gran influencia que puede llegar a tener la ley de Amdahl [2] sobre un sistema con partes concurrentes.

Primero que nada, es claro que, en el caso particular de este algoritmo, las ventajas de los sistemas de memoria compartida son significativas, ya que evitan el tener que reinsertar los registros después de ser modificados. A pesar de la posibilidad de condiciones de carrera, realmente no es un problema tan grande en este algoritmo, ya que si se dividen adecuadamente las partes sobre las que se va a procesar, no debería haber ningún problema.

Considero, también, que la implementación paralela de este algoritmo, en el caso de `Java` y `Erlang`, resulta muy sencilla de implementar ya que se tiene la versión secuencial, en particular por lo fácil que es expresarlos con un ciclo sencillo o con una función recursiva. Cualquiera de estas dos estructuras son fáciles de representar en forma paralela en ciertos lenguajes.

De la misma forma, resulta claro que implementar concurrencia con `JavaScript` para este tipo de algoritmos, tiene una complejidad superior, ya que cada uno de los `WebWorkers` requiere recibir mucha información y regresar mucha información. Además, esta información se debe volver a agregar a la matriz original.

Este ultimo punto podría resultar un problema en `Erlang` de la misma manera, ya que son sistemas que funcionan por memoria compartida. Sin embargo, se facilita mucho por la naturaleza funcional pura del lenguaje, lo cual hace que realizar operaciones en listas y, por extensión, matrices, resulte más natural y directo.

Un aspecto importante a destacar en este caso, es que pese a la gran ventaja que tiene en este caso el uso de un sistema de memoria compartida, no es el caso para todos los algoritmos con implementaciones paralelas. Me parece que en el caso de la inversión de matrices la gran ventaja se debe a las actualizaciones que deben ser realizadas sobre la matriz original.

Creo que uno de los aspectos más importantes que puedo rescatar de realizar este artículo y la implementación del algoritmo es que, como en cualquier otro paradigma de programación, existen múltiples herramientas, cada una con respectivas ventajas y desventajas y resulta importante reconocer las situaciones en las que unas pueden resultar superiores a otras, ya sea en términos de rendimiento o de implementación.

También opino que la decisión de eficiencia contra facilidad de implementación es muy dependiente de un proyecto a otro, aunque en general me parece que son pocas las situaciones en las que se pueda requerir un aumento lo suficientemente significativo en la eficiencia de un programa para que pueda considerarse como el factor principal. Sin embargo, sé que existen muchos sistemas sistemas, particularmente sistemas en los cuales llegan a depender vidas humanas, que requieren ser tan óptimos como sea posible.

Finalmente, puedo afirmar que resulta vital entender y saber realizar implementaciones de algoritmos en paralelo, ya que es un hecho que el aumento de velocidad es significativo y es, por las nuevas tendencias de los procesadores, la única forma de aprovechar los nuevos recursos que se ponen a nuestro alcance como desarrolladores de software. Creo que sería excelente incorporar el curso de programación multinúcleo al programa de estudios básico.

5. Agradecimientos

Agradecimiento especial a Gerardo Galíndez por el apoyo en la implementación del algoritmo con Erlang, principalmente en cuanto a la parte conceptual y alternativas para la implementación pensada.

Agradezco de igual forma al equipo completo de ShareLaTeX [3], por su herramienta en línea para realizar este artículo con facilidad y al sitio de la universidad de Carnegie Mellon [4] por el ejemplo de implementación de inversión de matrices en un lenguaje funcional.

Mención especial a los sitios de Wikibooks LaTeX[1], Texblog [2] y Stack Exchange [3] por la documentación para el uso correcto de LaTeXy mejor redacción del presente artículo.

Finalmente, agradecimientos a Saúl de Nova Caballero y a Eduardo Rodriguez Ruiz por la atención al exponer mis ideas para aclarar la implementación del algoritmo de inversión de matrices.

Notas

[1] https://en.wikibooks.org/wiki/LaTeX/Mathematics

[2] http://texblog.org/2014/06/24/big-o-and-related-notations-in-latex/

[3] http://tex.stackexchange.com/questions/40280/how-can-i-visualize-matrix-operations Accedido el 5 de noviembre del 2015

Referencias

[1] Indian Statistical Institute. *Matrix algorithms (part 1).*
http://www.isical.ac.in/ arnabc/matalgop1.pdf Accedido el 5 de noviembre del 2015.

[2] ACM Digital Library *Improvements in multiprocessor system design.*
http://doi.acm.org/10.1145/327070.327215 Accedido el 5 de noviembre del 2015

[3] ShareLaTeX. *About Us*
https://www.sharelatex.com/about Accedido el 5 de noviembre del 2015.

[4] Carnegie Mellon University *Matrix Inverse*
http://www.cs.cmu.edu/~scandal/nesl/alg-numerical.html#inverse Accedido el 5 de noviembre del 2015.

6. Apéndices

Código `HTML` para mostrar los resultados de la implementación de `JavaScript`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>Inversion Matrices</title>
    <link rel="stylesheet" href="style.css" />
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
    <script src="inverse.js"></script>
    <script src="setup_worker.js" type="javascript/worker"></script>
    <script src="pivot_worker.js" type="javascript/worker"></script>
  </head>

  <body>
    <h1>Inversion de Matrices</h1>
    <p> Tiempo Concurrente: <span id="parallel">?</span></p>
    <p> Tiempo Sequencial: <span id="sequential">?</span></p>
  </body>
</html>
```

Código de `JavaScript` para coordinar a los `WebWorkers` y para realizar los cálculos de la inversión de matrices:

```
'use strict';

var MAXN = 150;
var numWorkers = 4;
var startTime;

var original = [];
original.length = MAXN;

var padWorkers = [];
padWorkers.length = numWorkers;

var gaussWorkers = [];
gaussWorkers.length = numWorkers;

function gaussJordan(A, i) {
  if (i === A.length) {
    var endTime = performance.now();
    if (numWorkers > 1) {
      $("#parallel").html((endTime - startTime) / 1000);

      numWorkers = 1;
      padMatrix(original);
    } else {
      $("#sequential").html((endTime - startTime) / 1000);
    }
    return;
  }

  var irow = A.shift();
  var val = irow[i];

  var workersFinished = 0;
```

```
  var start = 0;
  var chunkSize = MAXN * 2 / numWorkers;
  var end;

  for (var j = 0; j < numWorkers; j++) {
    end = j + 1 === numWorkers ? MAXN - 1 : Math.ceil(start + chunkSize);
    gaussWorkers[j].postMessage({start: start, end: end, irow: irow, val: val});
    start = end;

    gaussWorkers[j].onmessage = function (event) {
      workersFinished++;
      for (var k = event.data.start; k < event.data.end; k++) {
        irow[k] = event.data.irow[k];
      }

      if (workersFinished === numWorkers) {
        workersFinished = 0;
        start = 0;
        chunkSize = (MAXN - 1) / numWorkers;

        for (var k = 0; k < numWorkers; k++) {
          end = k + 1 === numWorkers ? MAXN - 1 : Math.ceil(start + chunkSize);
          gaussWorkers[k].postMessage({start: start, end: end, irow: irow, i: i, A: A, size: MAXN});
          start = end;

          gaussWorkers[k].onmessage = function (event) {
            workersFinished++;
            for (var l = event.data.start; l < event.data.end; l++) {
              A[l] = event.data.jrow[l];
            }
            if (workersFinished === numWorkers) {
              A.push(irow);
              gaussJordan(A, i + 1);
            }
          };
        }
      }
    }
  };
}

function padMatrix(matrix) {
  startTime = performance.now();
  var workersFinished = 0;
  var start = 0;
  var chunkSize = MAXN / numWorkers;
  var end;

  for (var i = 0; i < numWorkers; i++) {
    end = i + 1 === numWorkers ? MAXN - 1 : Math.ceil(start + chunkSize);
    padWorkers[i].postMessage({start: start, end: end, matrix: matrix, size: MAXN});
    start = end;

    padWorkers[i].onmessage = function (event) {
```

```
      workersFinished++;
      for (var j = event.data.start; j < event.data.end; j++) {
        matrix[j] = event.data.padded[j];
      }
      if (workersFinished === numWorkers) {
        gaussJordan(matrix, 0);
      }
    };
  }
}

function printMatrix(matrix) {
  var res = "";
  for (var i = 0; i < MAXN; i++) {
    for (var j = MAXN; j < MAXN * 2; j++) {
      res += matrix[i][j] + ", ";
    }
    res += "\n";
  }
  console.log(res);
}

$(function () {
  var randomNum;
  for (var i = 0; i < MAXN; i++) {
    original[i] = [];
    original[i].length = MAXN;
    for (var j = 0; j < MAXN; j++) {
      randomNum = 1.0 + (Math.random() * MAXN);
      original[i][j] = randomNum;
    }
  }

  for (var i = 0; i < numWorkers; i++) {
    padWorkers[i] = new Worker('pad_worker.js');
    gaussWorkers[i] = new Worker('gauss_worker.js');
  }

  padMatrix(original.slice());
});
```

`WebWorker` de `JavaScript` para crear la matriz de identidad del tamaño deseado y agregarla a la matriz base:

```
onmessage = function(event) {
  var start = event.data.start;
  var end = event.data.end;
  var matrix = event.data.matrix;
  var size = event.data.size;

  for (var i = start; i < end; i++) {
    matrix[i].length = size * 2;
    for (var j = size; j < size * 2; j++) {
      matrix[i][j] = 0;
      if (size + i === j) {
```

```
        matrix[i][j] = 1;
      }
    }
  }

  postMessage({start: start, end: end, padded: matrix});
}
```

`WebWorker` de `JavaScript` para escalar una fila de la matriz o para hacer la sustitución hacia atrás de los valores encontrados:

```
onmessage = function(event) {
  var start = event.data.start;
  var end = event.data.end;
  var irow = event.data.irow;
  var val = event.data.val;
  var A = event.data.A;

  if (!A) {
    for (var i = start; i < end; i++) {
      irow[i] /= val;
    }

    postMessage({start: start, end: end, irow: irow});
  } else {
    var i = event.data.i;
    var size = event.data.size;
    for (var j = start; j < end; j++) {
      var jrow = A[j];
      var scale = jrow[i];
      for (var k = 0; k < size * 2; k++) {
        jrow[k] -= (irow[k] * scale * 1.0);
      }
    }
    postMessage({start: start, end: end, jrow: A});
  }
}
```

Código para la implementación de la inversión de matrices en `Java` usando `Streams`:

```
import java.util.Arrays;
import java.util.stream.*;

public class Inverse {
    public static int MAXN = 1000;

    // Sequential code
    public static double[][] invert(double matrix[][]) {
        for (int i = 0; i < MAXN; i++) {
            matrix[i][MAXN + i] = 1;
        }

        return gaussJordan(matrix);
    }
```

```
public static double[][] gaussJordan(double matrix[][]) {
    int i, j, k;
    double val, scale;
    double[] irow, jrow;

    for (i = 0; i < MAXN; i++) {
        irow = Arrays.copyOf(matrix[0], MAXN * 2);
        val = irow[i];
        for (j = 0; j < MAXN * 2; j++) {
            irow[j] /= val;
        }

        for (j = 1; j < MAXN; j++) {
            jrow = matrix[j];
            scale = matrix[j][i];
            for (k = 0; k < MAXN * 2; k++) {
                jrow[k] -= (irow[k] * scale);
            }
        }
        for (j = 0; j < MAXN - 1; j++) {
            matrix[j] = matrix[j + 1];
        }
        matrix[MAXN - 1] = Arrays.copyOf(irow, MAXN * 2);
    }
    return matrix;
}

// Parallel code
public static double[][] parallelInvert(double matrix[][]) {
    IntStream
        .range(0, MAXN)
        .parallel()
        .forEach(i -> {
            matrix[i][MAXN + i] = 1;
        });

    parallelGaussJordan(matrix);

    return matrix;
}

public static double[][] parallelGaussJordan(double matrix[][]) {
    IntStream.range(0, MAXN).forEach(i -> {
        double[] irow = Arrays.copyOf(matrix[0], MAXN * 2);
        double val = irow[i];

        IntStream.range(0, MAXN * 2).parallel().forEach(j -> {
            irow[j] /= val;
        });

        IntStream.range(1, MAXN).parallel().forEach(j -> {
            double[] jrow = matrix[j];
            double scale = matrix[j][i];
```

```
            for (int k = 0; k < MAXN * 2; k++) {
                jrow[k] -= (irow[k] * scale);
            }
        });

        IntStream.range(0, MAXN - 1).forEach(j -> {
            matrix[j] = matrix[j + 1];
        });

        matrix[MAXN - 1] = Arrays.copyOf(irow, MAXN * 2);
    });
    return matrix;
}

public static void printMatrix(double[][] matrix) {
    for (int i = 0; i < MAXN; i++) {
        for (int j = MAXN; j < MAXN * 2; j++) {
            System.out.print(matrix[i][j] + ", ");
        }
        System.out.println();
    }
    System.out.println();
}

public static void benchmark(String title, double[][] matrix, boolean sequential) {
    System.out.println(title);

    long start = System.currentTimeMillis();
    if (sequential) {
        invert(matrix);
    } else {
        parallelInvert(matrix);
    }
    long end = System.currentTimeMillis();
    double time = (end - start) / 1000.0;

    System.out.printf("Time: %.2f%n", time);
}

public static void main(String... args) {
    double randomNum;
    double[][] sequential = new double[MAXN][MAXN * 2];
    double[][] parallel = new double[MAXN][MAXN * 2];
    for (int i = 0; i < MAXN; i++) {
        for (int j = 0; j < MAXN; j++) {
            randomNum = 1.0 + (Math.random() * MAXN);
            sequential[i][j] = randomNum;
            parallel[i][j] = randomNum;
        }
    }

    benchmark("Sequential", sequential, true);
    benchmark("Concurrent", parallel, false);
}
```

```
}
```

Código para la implementación de la inversión de matrices en `Erlang`:

```
-module(inverse).
-compile(export_all).

%start Sequential Implementation

%start Sequential padding

sequential_row(1) -> [0];
sequential_row(I) -> [0 | sequential_row(I - 1)].

sequential_identity(S, S) -> [];
sequential_identity(S, R) ->
    {A, B} = lists:split(R, sequential_row(S - 1)),
    I = [A ++ [1] ++ B],
    I ++ sequential_identity(S, R + 1).

sequential_pad(A) ->
    I = sequential_identity(length(A), 0),
    sequential_pad(A, I, length(A)).

sequential_pad(_, _, 0) -> [];
sequential_pad(A, I, R) ->
    Row = [lists:nth(R, A) ++ lists:nth(R, I)],
    sequential_pad(A, I, R - 1) ++ Row.

%end Sequential padding

%start Sequential GaussJordan

sequential_scale([], _, _) -> [];

sequential_scale([V | J], [X | I], S) ->
    R = V - (S * X),
    [R] ++ sequential_scale(J, I, S).

sequential_drop_identity(A) ->
    N = length(A),
    lists:map(fun(R) ->
                      {_, X} = lists:split(N, R),
                      X
              end, A).

sequential_gauss_jordan(A, I) ->
    case (I > length(A)) of
        true -> sequential_drop_identity(A);
        _ ->
            [Ir | T] = A,
            V = lists:nth(I, Ir),
            Irow = lists:map(fun(X) -> X / V end, Ir),
            Ap = lists:map(fun(Jrow) -> sequential_scale(Jrow, Irow, lists:nth(I, Jrow)) end, T),
```

```
            sequential_gauss_jordan(Ap ++ [Irow], I + 1)
    end.

%end Sequential GaussJordan

sequential_inverse(A) ->
    Ap = sequential_pad(A),
    sequential_gauss_jordan(Ap, 1).

%end Sequential Implementation

%start Concurrent Implementation

spawn_processes(0, _Function, _Args, _Message, _ParentId) -> ok;
spawn_processes(N, Function, Args, Message, ParentId) ->
    Pid = spawn(inverse, Function, Args),
    Pid ! {Message, ParentId},
    spawn_processes(N - 1, Function, Args, Message, ParentId).

%start Concurrent Padding

concurrent_row(1) -> [0];
concurrent_row(I) ->
    R = [0 | sequential_row(I - 1)],
    receive
        {compute, Pid} -> Pid ! {result, R}
    end.

concurrent_identity(S, R) ->
    spawn(inverse, spawn_processes, [S, concurrent_row, [S - 1], compute, self()]),
    concurrent_identity(S, R, spawned).

concurrent_identity(S, S, spawned) -> [];
concurrent_identity(S, R, spawned) ->
    receive
        {result, Row} ->
            {A, B} = lists:split(R, Row),
            I = [A ++ [1] ++ B],
            I ++ concurrent_identity(S, R + 1, spawned)
    end.

concurrent_pad(A) ->
    I = concurrent_identity(length(A), 0),
    sequential_pad(A, I, length(A)).

%end Concurrent Padding

%start Concurrent GaussJordan

concurrent_drop_identity(A) ->
    N = length(A),
    plists:map(fun(R) ->
                       {_, X} = lists:split(N, R),
                       X
```

```
                end, A).

concurrent_gauss_jordan(A, I) ->
    case (I > length(A)) of
        true -> concurrent_drop_identity(A);
        _ ->
            [Ir | T] = A,
            V = lists:nth(I, Ir),
            Irow = plists:map(fun(X) -> X / V end, Ir),
            Ap = plists:map(fun(Jrow) -> sequential_scale(Jrow, Irow, lists:nth(I, Jrow)) end, T),
            concurrent_gauss_jordan(Ap ++ [Irow], I + 1)
    end.

%end Concurrent GaussJordan

concurrent_inverse(A) ->
    Ap = concurrent_pad(A),
    concurrent_gauss_jordan(Ap, 1).

%end Concurrent Implementation

%start Benchmarks

generate_row(0, _) -> [];
generate_row(N, MAXN) -> [random:uniform(MAXN) | generate_row(N - 1, MAXN)].

generate_matrix(0, _) -> [];
generate_matrix(N, MAXN) -> [generate_row(MAXN, MAXN) | generate_matrix(N - 1, MAXN)].

benchmark(MAXN) ->
    Mat = generate_matrix(MAXN, MAXN),
    %[[1.0, 1.0, 2.0, 1.0], [1.0, 2.0, 1.0, 2.0], [1.0, 1.0, 1.0, 3.0], [2.0, 3.0, 1.0, 3.0]],
    T1 = erlang:system_time(milli_seconds),
    sequential_inverse(Mat),
    T2 = erlang:system_time(milli_seconds),
    TS = (T2 - T1) / 1000,
    T3 = erlang:system_time(milli_seconds),
    concurrent_inverse(Mat),
    T4 = erlang:system_time(milli_seconds),
    TC = (T4 - T3) / 1000,
    io:format("Sequential time: ~w~n", [TS]),
    io:format("Concurrent time: ~w~n", [TC]),
    io:format("Speedup: ~w~n", [TS/TC]).

%end Benchmarks
```