Paralelización de la Factorización LU Usando el Lenguaje ZPL

Parallelization of the LU Decomposition with the ZPL Programming Language

Jorge Castellanos (*) José Luis Ramírez (*) Demetrio Rey (**)
(*) Facultad de Ciencias y Tecnología. (**) Instituto de Matemáticas y Cálculo Aplicado, Facultad de Ingeniería. Universidad de Carabobo. Valencia, Venezuela
jcasteld@uc.edu.ve jbarrios@uc.edu.ve drey@uc.edu.ve

Resumen

En este trabajo se presenta una sencilla implementación de la Factorización LU usando el Lenguaje de Programación Paralela ZPL. La implementación presentada aprovecha las características más importantes de ZPL: simplicidad, portabilidad y ejecución eficiente en ambientes paralelos. Se describe el programa ZPL para la Factorización LU y se compara su rendimiento paralelo en un Cluster tipo Beowulf versus una implementación equivalente con ScalaPack sobre C/MPI. Se demuestra que el código ZPL es sustancialmente mas corto que el de Scalapack y además alcanza un rendimiento comparable a éste (de menos de un orden de magnitud).

Palabras Clave: Factorización LU, ZPL, Paralelismo, Scalapack

Abstract

We present a concise implementation of the LU Decomposition using the High-Level Parallel Programming Language ZPL. Our implementation takes advantage of the most important features of ZPL: simplicity, portability and efficient execution on parallel environments. We describe our ZPL program for the LU decomposition and compare its parallel performance on a Beowulf cluster versus an equivalent implementation with ScalaPack on C/MPI. We show that the ZPL source code is substantially shorter than ScalaPack code, and also exhibits performance competitive with ScalaPack (within one order of magnitude).

Index Terms: LU Decomposition, ZPL, Parallelism, ScalaPack

1. Introducción

El método *directo* comúnmente utilizado para resolver sistemas de ecuaciones lineales; en términos modernos se llama *descomposición LU con pivoteo*, ó *factorización LU con pivoteo*, tal como lo cita [1]. En detalle: dada A una matriz nxn, no singular. La eliminación gaussiana con pivoteo parcial da la factorización $PA = LU$, como es mostrado en [3], donde P es una matriz de permutación, L es una matriz triangular inferior y U es una matriz triangular superior.

La solución de $Ax = b$ se convierte en $LUx = Pb$, dado que $PAx = Pb$. La solución de $LUx = Pb$ puede hacerse en tres pasos:

1. Establecer $d = Pb$, mezclando entradas de b, o accediendo por direccionamiento indirecto un vector de permutación.
2. Resolver $Ly = d$ (sistema triangular inferior)
3. Resolver $Ux = y$ (sistema triangular superior)

Este documento presentamos la implementación de la factorización LU en el lenguaje de programación paralela ZPL. En la Sección 1 se hace una breve introducción a la factorización *LU*, en la Sección 2 se presenta una breve introducción a los algoritmos de factorización *LU* y se muestra una versión modificada del algoritmo que facilita su paralelización. En la Sección 3 se presenta una introducción al lenguaje ZPL con el objeto de facilitar la comprensión de la implementación del algoritmo paralelo. En la Sección 4 se presenta y comenta la implementación del algoritmo en el lenguaje ZPL. En la Sección 5 se presenta una versión equivalente en ScalaPack. En la Sección 6 se discute la experimentación realizada con ambas versiones. En la Sección 7 se presentan los resultados y en la Sección 8 se discuten los mismos. Por último, en la Sección 9 se resumen las conclusiones obtenidas y en la Sección 10 las recomendaciones pertinentes para futuros trabajos en este tema.

2. La Factorización LU

2.1. Algoritmo básico

El algoritmo clásico de factorización *LU* sigue una secuencia de tres fases en cada paso: búsqueda del pivote, intercambio de filas para traer el pivote a la posición diagonal, escalado de las entradas subdiagonales en la columna, luego se actualiza el resto de la matriz. Al realizar estas operaciones sobrescribiendo la matriz *A* se obtienen los factores *L* y *U* en las partes correspondientes de *A* (puesto que *L* es una triangular inferior unidad, no se almacena su diagonal, así que todo cabe limpiamente). El algoritmo básico sin pivoteo sigue una secuencia de dos fases en cada paso: escalado de las entradas subdiagonales en cada columna y actualización del resto de la matriz *A*. Usando pseudo-notación de MATLAB® el algoritmos básico se muestra en la Figura 1.

```
for k= 1:n-1
  for i= k+1:n % escalado
      A(i,k) = A(i,k)/A(k,k)
  end for
  for i= k+1:n % actualización
      for j= k+1:n
        A(i,j) = A(i,j) - A(i,k)*
                          A(k,j)
      end for
  end for
end for
```

Fig. 1. Factorización LU: algoritmo básico

2.2. La Factorización LU con actualización rango-1

Usando la notación Matlab descrita anteriormente [2], el algoritmo de la Figura 1 se puede indicar en forma más compacta como se muestra en la Figura 2, tal como es mostrado por [2].

```
for k = 1:n-1
  % Primer for (i= k+1:n)
  A(k+1:n,k) = A(k+1:n,k)/A(k,k)
  % Segundo for (i= k+1:n)
  A(k+1:n,k+1:n) = A(k+1:n,k+1:n) -
      A(k+1:n,k)*    A(k,k+1:n)
end for
```

Fig. 2. Factorización *LU* con actualización rango-1

De hecho, podemos ajustarnos más a la notación de Matlab usando vectores indexados ($K = k+1:n$ crea un vector K con elementos enteros entre $k+1$ y n). En la Figura 3 se muestra el código de la Figura 2 usando el vector indexado *K*.

```
for k = 1:n-1
  K = k+1:n
  % escalamiento
  A(K,k) = A(K,k)/A(k,k)
  % actualización rango-1
  A(K,K) = A(K,K) - A(K,k)* A(k,K)
end for
```

Fig. 3. Factorización *LU* con actualización rango-1 usando vector indexado (*K*)

En cualquiera de las dos formas pasadas (Figura 2 y Figura 3), se muestra más claramente cuáles son las operaciones involucradas (escalamiento y actualización). Cada una de las iteraciones contiene un escalamiento del vector en la parte sub-diagonal de la columna *k*, seguida por una actualización rango-1 de la parte restante de la matriz. En la Figura 4 se muestra gráficamente el procedimiento con la actualización del rango-1 en la factorización *LU*.

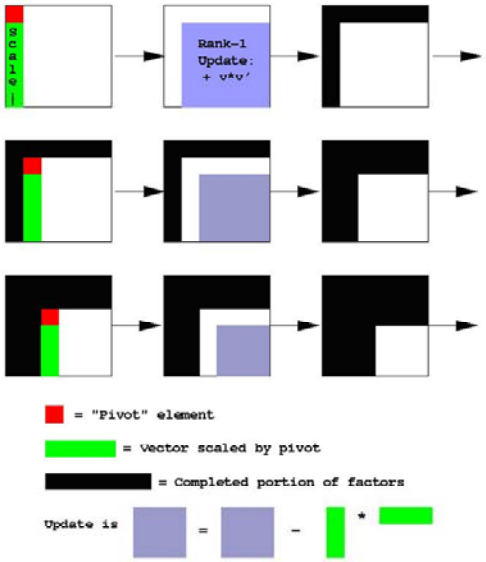

Fig. 4. Factorización LU con actualización rango-1 (Fuente: [2])

2.3. Matrices de permutación

Las matrices de permutación nunca se almacenan o manipulan como matrices [6]. En su lugar hay dos representaciones que utilizan solamente un vector entero. Una cosa es utilizar un vector de permutación y la otra es utilizar un vector pivote. El primero se almacena en *P* como un conjunto de números enteros p_i que representan la posición de x_i en $y = Px$:

```
I  :   1   2   3   4   5   6   7   8
p_i :  3   7   5   8   4   1   2   6
```

Fig. 5. Vector de permutación

La matriz de permutación correspondiente es entonces:

```
     [0  0  1  0  0  0  0  0]
     |0  0  0  0  0  0  1  0|
     |0  0  0  0  1  0  0  0|
P=   |0  0  0  0  0  0  0  1|
     |0  0  0  1  0  0  0  0|
     |1  0  0  0  0  0  0  0|
     |0  1  0  0  0  0  0  0|
     [0  0  0  0  0  1  0  0]
```

Fig. 6. Matriz de permutación

y nadie claramente representaría esta matriz usando un arreglo de $n^2 = 64$ elementos. Tal vez resulte conveniente para una matriz de 8x8, pero para matrices de 2000 x 2000, no se recomienda definitivamente.

La segunda manera de representar las matrices de permutación *P* es con vectores pivote, y es en éstos donde utilizamos la eliminación Gaussiana. Este es un arreglo entero *piv* de longitud *n*, aplicado a un vector *x* en $y = Px$.

2.4. La Factorización LU con pivoteo

En [4] se establece que, al usar la versión anterior (actualización rango-1) de la factorización *LU*, la división por *A(k,k)* en la operación del escalamiento puede causar problemas si *A(k,k)* es un número pequeño en valor absoluto. Pivoteo significa traer al elemento más grande en esa posición intercambiando filas o columnas en la matriz. Pivoteo parcial significa hacer solamente intercambio de filas; la factorización *LU* es "generalmente estable" cuando se utiliza el pivoteo parcial. Sin embargo, hay clases de problemas (algunos problemas de Ecuaciones Diferenciales Ordinarias con dos puntos de valores de frontera son un ejemplo) para los cuales él método puede fallar. Agregándole pivoteo a la versión rango-1 tenemos el algoritmo de la Figura 7.

```
for k = 1:n
     p = index del  máximo elemento en
                              |A(k:n,k)|
     piv(k) = p
     intercambiar filas k y p de A:
                         A(k,:)<--->A(p,:)
     A(k+1:n,k) = A(k+1:n,k)/A(k,k)
     A(k+1:n,k+1:n) = A(k+1:n,k+1:n) -
                   A(k+1:n,k)* A(k,k+1:n)
end for
```

Fig. 7. Versión con actualización rango-1 y pivoteo parcial

Obsérvese en la Figura 8 que el primer paso selecciona el elemento más grande en la *k*-ésima columna entre la parte restante (no reducida) de la matriz. También, el índice del lazo itera desde 1 hasta *n*, de modo que también será definido *piv(n)* – que es por supuesto *n*. También, obsérvese que se intercambian las filas enteras *k* y *p* del arreglo *A*; esto también intercambiará las partes correspondientes de *L* y *U* puesto que estamos sobrescribiendo *A* con esos factores. Esta es otra característica interesante de este procedimiento de factorización *LU* el cual hace que los factores salgan en el "orden correcto".

```
y = x
for k = 1:n
   intercambiar y(k) y y(piv(k)) en el
                vector y
end for
```

Fig. 8. Uso del vector pivote

3. Una breve introducción a ZPL

ZPL es un lenguaje de programación basado en arreglos [6], con paralelismo de datos utilizado para programar computadoras paralelas cuando se desea un alto rendimiento incluso si el tiempo de desarrollo es limitado. Su relativa simplicidad y sensación semejante a Pascal lo hacen fácil de leer y entender, conservando aún un modelo sofisticado de paralelismo. Por estas razones y porque se han implementado reducciones basadas en arreglos, elegimos implementar nuestro algoritmo en el contexto de ZPL. Se debe observar, sin embargo, que el algoritmo planteado es suficientemente general para aplicarlo en otros lenguajes paralelos de alto nivel. En esta sección introducimos las características del lenguaje ZPL relevantes para este trabajo. Los lectores interesados en profundizar pueden referirse a la guía del usuario [5].

3.1. Regiones y arreglos

El concepto principal de ZPL es la región, las cuales son un conjunto de índices sin datos asociados. Para declarar dos regiones, R y BigR, donde BigR es un conjunto de *(n+2) x (n+2)* índices, mientras que R es un conjunto de *n x n* índices que se refiere solamente a la porción sin bordes de BigR. Esta declaración se realiza de la siguiente manera:

```
region  BigR = [0..n+1, 0..n+1];
        R = [1..n, 1..n];
```

Las regiones se utilizan en dos contextos. Primero se usan para declarar arreglos paralelos. Todos los

arreglos declarados sobre regiones son paralelos y como tales, son distribuidos sobre los procesos de forma especificable en tiempo de ejecución. Se declaran tres arreglos enteros A, B y C definidos sobre el conjunto de índices dado por BigR:

```
var A, B, C : [BigR] integer;
```

El segundo uso de las regiones es para señalar implícitamente el cálculo paralelo. Por ejemplo para sumar los valores correspondientes a la porción sin bordes de la región de los arreglos A y B, y almacenar el resultado en el arreglo C, se escribe la siguiente línea de código:

```
[R] C := A + B;
```

Esta línea se corresponde con un doble ciclo anidado sobre el conjunto de índices *n x n*. Obsérvese como el operador + se aplica a cada elemento de A y B. De esta forma, + es un operador aritmético "elemento-a-elemento", al igual que otros operadores tradicionales: -, *, /; que definen la resta, multiplicación y división elemento a elemento, respectivamente.

ZPL posee otros operadores de arreglos que permiten efectuar complejas operaciones sobre vectores y matrices con un número reducido de instrucciones. En esta sección, además de los operadores básicos elemento a elemento ya mencionados, sólo nos referiremos a aquellos que fueron utilizados en la factorización LU: *flood* y *reduce*.

3.2. Flood (replicación)

Una de las características poderosas de ZPL [7], a objeto de aprovechar la utilización de múltiples procesadores es la habilidad de llenar una matriz con copias de una fila o columna, o más generalmente, el llenar un arreglo de dimensión mayor con copias de otro arreglo de dimensión menor. Esta operación, llamada *flood* en ZPL (simbolizada como ">>") es una generalización de la idea de la promoción escalar. Para describir el *flood*, se introducirá un cierto vocabulario básico:

Dimensión flood: Una dimensión en una especificación de región en la cual se reemplaza el rango por un asterisco. Por ejemplo, la segunda dimensión en la especificación de región `C = [1..n,*]` es una dimensión *flood*.

Región flood: Es una región definida con uno o más dimensiones *flood*. Por ejemplo, la región `F = [*,1..n]` es una región *flood* con una primera dimensión *flood*.

Arreglo flood: es un arreglo declarado sobre una región flood. Por ejemplo:

```
var Flrows: [F] float;
    Flcols: [C] float;
```

declara dos arreglos *flood*, asumiendo las regiones F y C de las definiciones anteriores.

El concepto detrás del *flood* es que las dimensiones especificadas, por ejemplo, las dimensiones no-*flood*s, definen el tamaño de los elementos que son replicados. El asterisco especificando una dimensión *flood* puede ser leído como "un número indeterminado de", tal como en "El arreglo `Flrows` contiene un indeterminado número de filas iguales de n elementos y el arreglo `Flcols` contiene un indeterminado número de columnas iguales de n elementos".

A continuación se muestra cómo efectuar la replicación de la k-ésima columna de A en toda la matriz B:

```
[1..n, *] B := >>[1..n, k] A
```

Obsérvese como >> necesita dos regiones: una región fuente `[1..n, k]`, localizada a la derecha del operador >>, y una región destino `[1..n,*]`, a la izquierda de la asignación. La región fuente siempre tiene una dimensión colapsada (un solo índice) y la región destino, en la misma dimensión colapsada de la fuente, tendrá generalmente una dimensión *flood* o en su defecto una especificación de rango `a..b`.

En la práctica, el operador >> resulta ser de frecuente uso en muchos de los algoritmos típicos implementados en ZPL, entre ellos la factorización LU.

3.3 Reduce (reducción)

Reduce (<<) puede verse en cierta manera como lo contrario de un *flood*, en vez de replicar elementos para llenar arreglos de mayor orden, << reduce los arreglos para obtener arreglos de menor orden, o incluso escalares (de allí su nombre: "reducir"). El operador << precisa de otro operador aritmético (asociativo y conmutativo) para llevar a cabo las reducciones. Por ejemplo, para obtener el máximo valor presente en un vector V de n elementos, aplicamos un *max-reduce* (max<<)

```
[1..n]    m := max<< V
```

Donde *m* es un escalar al cual se le asigna el máximo valor de *V*. Si necesitamos obtener la suma

total de los elementos de la matriz *A* de *n x n*, basta con aplicar un +*reduce*:

```
[1..n, 1..n ] suma := +<< A
```

Por su versatilidad y facilidad de uso, el operador de reducción <<, al igual que el *flood*, resulta ser de frecuente uso en muchos de los algoritmos implementados en ZPL. Igualmente éste es empleado en la factorización LU.

3.4. Modelo de parelización y costo de comunicación de operadores

ZPL paraleliza automáticamente las operaciones sobre sus arreglos. No existen llamadas a funciones o directivas de paralelismo. La estrategia de división de tareas del lenguaje consiste en paralelismo de datos: todos los procesadores efectúan la misma secuencia de instrucciones sobre el pedazo de arreglo asignado a cada procesador.

La comunicación entre procesadores sólo sucede cuando se usan ciertos operadores de ZPL. Dado que ZPL distribuye la "superregión" (la unión de todas las regiones) de un determinado problema en forma equitativa entre todos los procesadores, todos los arreglos son automáticamente alineados y distribuidos de la misma manera. De allí que en ZPL se cumpla la afirmación "mismo índice = mismo procesador". Así, por ejemplo, el elemento A[3,2] estará en el mismo procesador que el elemento B[3,2] o C[3,2], y asi con cualquier arreglo del problema que esté definido en la coordenada (3,2).

Este modelo de particionamiento implica que las operaciones elemento a elemento no inducen comunicación alguna, porque no producen movimiento de elementos entre índices. Mientras que operaciones tal como *flood* y *reduce*, si provocarán comunicación, dado que son operaciones globales que requieren de datos provenientes de diversos índices.

Este sencillo modelo de paralelización permite al programador ZPL predecir hasta cierto punto qué tan bien correrá su algoritmo en una máquina paralela, porque se tiene una cierta idea a priori del costo de comunicación de cada operador y el tamaño de la región sobre la cual opera.

4. Algoritmo de factorización *LU* en ZPL

La programación de una aplicación paralela en ZPL consta de dos partes:

- Configuración y definición de variables y regiones.
- Codificación del algoritmo haciendo uso de los operadores que hagan un uso más eficiente de la máquina paralela.

4.1 Configuración y definición de variables y regiones

La configuración y definición de variables puede observarse en la Figura 9 y consta de tres secciones: las variables de configuración (config var), las cuales se pueden modificar en el momento de invocar la ejecución del programa. Para este caso se definieron como variables de configuración: el orden la matriz cuadrada que se va a procesar, el nombre del archivo que contiene a la matriz de entrada y el nombre del archivo para la matriz de salida.

```
config var
--Número de filas y  columnas de la
matriz cuadrada A n x n
  n : integer = 3;

region
--región para la matriz n x n
  R  = [1..n,1..n];
--región flood para las filas
  Rf = [*,1..n];
--región flood para columnas
  Rc = [1..n,*];

var
--matriz sobre R
  A : [R] float;

--indices de iteración y pivoteo
  k,ipiv,ipiv2 : integer;

--arreglos flood para columna y fila
  Col : [Rc] float;
  Fila: [Rf] float;

--arreglos para intercambio de filas
  RowSwap1, RowSwap2 : [Rf] float;

-- escalares diagonal y máximo por col.
  diag, maximo : float;
```

Fig. 9. Definición de las variables

Las regiones son uno de los ingredientes más importantes en la confección del programa paralelo, ya que las mismas definen el nivel de paralelismo e implícitamente permiten al ZPL realizar la partición mas adecuada en función de la gris de ejecución y el hardware subyacente. En nuestro caso particular, se definieron tres regiones: la primera (R) conforma el espacio (filas y columnas) donde reside primariamente la matriz. Aprovechando la característica de flooding (ver 3.2) que tiene ZPL para el manejo paralelo de vectores y arreglos, se definieron dos secciones tipo *flood*, una para la manipulación de filas (Rf) y otra para el manejo de vectores columna (Rc). Las variables restantes hacen uso del concepto de las regiones y de los tipos básicos que vienen incorporados con el lenguaje; entre ellas

se pueden destacar: la variable A de tipo flotante definida en la región R que contiene la matriz a factorizar y luego en la misma queda la matriz factorizada después de terminar el proceso de factorización; los arreglos *flood* Col y Fila definidos en las regiones Rc y Rf respectivamente que permiten la manipulación de vectores filas y columna durante el proceso de factorización.

4.2. Código del algoritmo paralelo

El algoritmo de factorización se basa esencialmente en el mostrado en la Figura 8, el cual se desarrolla dentro de un ciclo que se repite "*n*-1" veces y que en su interior consta de cuatro partes:

1. Obtención del nuevo pivote (*p*), como el máximo elemento (en valor absoluto) de la submatrix que tiene por filas los índices que van desde *k* hasta *n* y por columna el índice *k*.
2. Intercambiar la fila *k* (actual) con la fila del nuevo pivote (*p*).
3. División (escalado) de la submatrix que tiene por filas los índices que van desde *k*+1 hasta n y por columna el índice *k* por el nuevo pivote (*p*).
4. Actualización (rango-1) de la submatrix definida por los índices de fila *k*+1:*n* y por los índices de columna *k*+1:*n*.

5. Algoritmo de factorización *LU* en ScalaPack

Para establecer una comparación en cuanto a la complejidad del código y el rendimiento de ejecución en una arquitectura paralela, se elaboró un programa equivalente de factorización LU usando la biblioteca numérica optimizada para sistemas paralelos ScalaPack. Por razones de espacio, se omite la presentación del código de esta versión.

6. Experimentación

6.1. Datos

Se emplearon matrices provenientes de pruebas industriales y científicas obtenidas de Matriz Market (http://math.nist.gov/MatrixMarket/), el cual es un repositorio de datos para pruebas en el estudio comparativo de algoritmos relacionados con el álgebra lineal numérica.

```
Procedure factorizacion();
[R] begin
        for k := 1 to n-1 do

         -- busqueda del mejor pivote
 [k..n,k]     begin
                 maximo := max<<fabs(A);
                 if( maximo=0.0 ) then
                   writeln("Fracasa
                         Factorización");
                   exit;
                 end;
                 ipiv := min<<(
                (fabs(A)!=maximo)*(n+1)+
               (fabs(A)=maximo)*(Index1)
                  );
              end;

         -- intercambio de filas
              if (ipiv != k) then
[*,]            RowSwap1 := >>[k,] A;
[*,]            RowSwap2 := >>[ipiv,] A;
[k,]            A := RowSwap2;
[ipiv,]         A := RowSwap1;
              end;

          -- escalado
[k,k]         diag := max<<A;
[k+1..n,k]    A := A/diag;

          -- actualización
[k+1..n,*]      Col := >>[k+1..n,k]A;
[*,k+1..n]      Fila := >>[k,k+1..n]A;
[k+1..n,k+1..n] A := A – Col * Fila;
          end;
end;
```

Fig 10. Implementación del algoritmo de factorización LU

6.2. Plataforma Computacional

Todas las pruebas se desarrollaron en el cluster NIMBUS de la Facultad de Ingeniería de la Universidad de Carabobo, el cual posee un total de 8 nodos, cada uno un Procesador Intel P4 de 2.4 Ghz, 1GB de RAM, con el Sistema Operativo Linux GNU/Debian e interconectados entre si mediante una red de 1 Gbit ethernet.

6.3. Casos de Prueba

A continuación se muestran las características principales de las matrices empleadas en la pruebas del algoritmo implementado. Estas matrices fueron tomadas de distintas áreas y de distintas aplicaciones con el fin de comprobar la generalidad del algoritmo implementado. En total, se tomaron un total de dos matrices, cuyas características principales se muestran en la siguiente tabla.

Tabla 1
Matrices de prueba

Nombre	Dimensión	Área
s1rmq4m1	5489x5489	Análisis de Elementos Finitos
dw8192	8192x8192	Guías de Onda Eléctrica

7. Resultados

En la Figura 11 se muestra una comparación en cuanto al número de líneas de código entre el algoritmo implementado en ZPL y el algoritmo implementado con la biblioteca numérica ScalaPack.

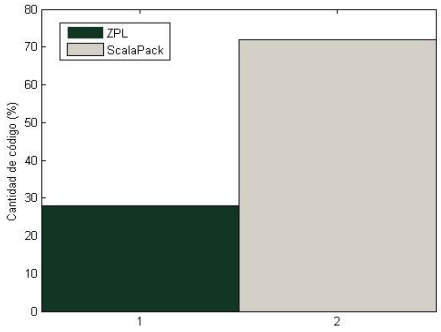

Fig. 11. Comparación de líneas de código entre el algoritmo de factorización *LU* en ZPL y ScalaPack

En las tablas 2 y 3 podemos observar el tiempo de ejecución, medido en segundos, del algoritmo implementado aplicado a las matrices descritas anteriormente, tomando una grid de *n x 1* procesadores, en estas mismas tablas se puede observar la aceleración y la eficiencia obtenida. En las Figuras 12, 13, se grafican los resultados mostrados de la Tabla 2. Por razones de espacio, se omiten las gráficas correspondientes la Tabla 3. Sin embargo, las mismas presentan un comportamiento similar a las Figuras de la Tabla 2, con la salvedad que ZPL mejora su rendimiento para esta matriz de mayor tamaño.

8. Discusión de los resultados

Las Figuras 12 y 13 evidencian un mejor rendimiento de la librería ScalaPack respecto a nuestra versión ZPL, lo cual era esperado por nosotros debido al carácter optimizado de la biblioteca matemática frente al característico *overhead* que presenta un lenguaje de programación paralelo tal como ZPL. Tanto ScalaPack como ZPL presentan una aceleración a medida que aumentan el número de procesadores utilizados, pero ScalaPack demuestra mejor rendimiento para cada uno de los casos. Se observa que ZPL escala mejor para la segunda matriz de mayor tamaño, lo cual es consistente con la reducción del costo de comunicación respecto al cómputo útil del algoritmo sobre una cantidad de datos mayor (ver aceleración y eficiencia de Tabla 3 respecto a Tabla 2).

Tabla 2
Matriz dw8192 (5489x5489)

Grid	Procesos	Tiempo ScalaPack	Tiempo ZPL	Aceleración ZPL	Eficiencia ZPL
1x1	1	73.04	341.48	1.00	1.000
2x1	2	50.72	265.74	1.29	0.643
3x1	3	35.84	243.43	1.40	0.468
4x1	4	27.18	160.35	2.13	0.532
5x1	5	21.27	175.68	1.94	0.389
6x1	6	17.65	174.24	1.96	0.327
7x1	7	14.89	152.31	2.24	0.320
8x1	8	13.16	94.29	3.62	0.453

Tabla 3
Matriz s1rmq4m1 (8192x8192)

Grid	Procesos	Tiempo ScalaPack	Tiempo ZPL	Aceleración ZPL	Eficiencia ZPL
1x1	1	201.79	1311.83	1.00	1.000
2x1	2	119.22	917.82	1.42	0.710
3x1	3	95.76	722.79	1.84	0.613
4x1	4	70.24	547.14	2.39	0.598
5x1	5	57.61	494.69	2.65	0.530
6x1	6	47.06	492.91	2.79	0.465
7x1	7	41.22	435.59	3.01	0.430
8x1	8	34.18	311.85	4.20	0.525

Sin embargo, aún así podemos cuantificar cómo el rendimiento de ZPL siempre se mantiene debajo de un orden de magnitud de diferencia (<10X) respecto a ScalaPack (tercera y cuarta columnas en Tablas 2 y 3), con la ventaja que el código ZPL para la factorización LU es bastante compacto en comparación con ScalaPack, tal como revela la substancial diferencia de líneas de código (Figura 11), y la total ausencia de directivas o llamadas a funciones de comunicación o sincronización.

Adicionalmente, para poder emplear la biblioteca ScalaPack, el programador requiere de conocimientos adicionales sobre el particionamiento de la matriz y la estructura de la puesta de ejecución, sin embargo, al

emplear el lenguaje de programación ZPL, estos detalles son ocultos para el programador.

La situación planteada anteriormente se repite para la otra matriz de prueba (Tabla 3) aunque para este caso, en comparación con el primero se puede notar que el comportamiento de la implementación con ZPL es mejor, tanto en linealidad como en mayor eficiencia.

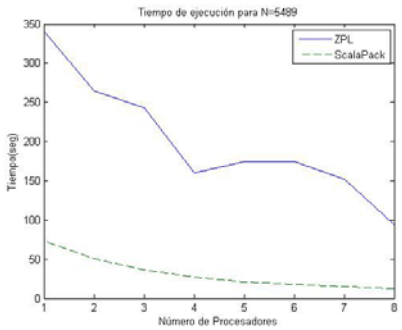

Fig. 12: Tiempo de Ejecución. Caso: Matriz s1rmq4m1

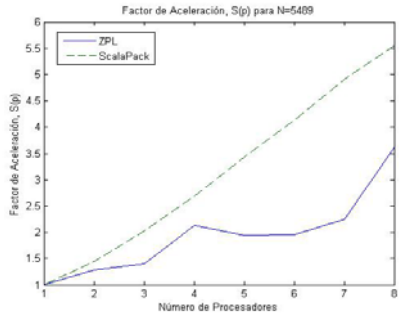

Fig. 13: Factor de Aceleración. Caso: Matriz s1rmq4m1

9. Conclusion

Se logró una primera versión paralela del algoritmo de factorización *LU* mediante un algoritmo sencillo de comprender definido usando los conceptos clave de regiones y los operadores paralelos del lenguaje de programación ZPL.

Si bien las curvas de aceleración obtenidas no muestran un comportamiento lineal en el cluster Nimbus, se realizaron pruebas (no mostradas en este trabajo) que mostraron un mejor comportamiento usando un cluster propietario marca Sun basado en procesadores dual Opteron.

Consideramos que la implementación cumplió el objetivo primordial de lograr reducir los tiempos de ejecución al generar un código paralelo sencillo y de fácil ejecución que delega al lenguaje de programación los problemas de configuración de hardware y particionamiento de datos.

Si bien el cluster Nimbus de 8 procesadores muestra disminución en los tiempos de ejecución hasta el octavo procesador, fue posible corroborar que había mejora en los tiempos de ejecución usando un mayor número de procesadores, lo cual muestra que la implementación hace un uso adecuado de la comunicación entre procesadores.

A medida que se aumenta el tamaño de la matriz, ZPL permite mejorar el desempeño en cuanto a aceleración y rendimiento, lo cual ya se comentó en la parte de resultados.

10. RECOMENDACIONES

Consideramos que el algoritmo de factorización *LU* se puede mejorar agregando técnicas como el producto matriz-vector como lo implementan las rutinas BLAS, sólo que debe considerarse para ello la utilización de operadores que tengan poca comunicación para no degradar el rendimiento del algoritmo en arquitecturas multiprocesador.

Creemos que el algoritmo paralelo desarrollado para ZPL puede probarse con otros lenguajes paralelos como HPF (High Performance Fortran) y evaluar el desempeño en cuanto a escalabilidad y rendimiento.

Falta evaluar el algoritmo usando un cluster con un mayor número de nodos, ya que las gráficas muestran una tendencia de incremento del rendimiento para un mayor número de nodos en el caso de matrices de gran tamaño.

REFERENCIAS

[1] Akai, Terrence J. *Métodos numéricos aplicados a la ingeniería*. Limusa Wiley. México 2004.

[2] Bramley, Ramdall. "Introduction to Scientific Computing. Class notes". Gaussian Elimination (LU Factorization) and the BLAS. Disponible: http://www.cs.indiana.edu/classes/p573/notes/arch/LA1.html.

[3] Burden, Richard L.; Faires, J. Douglas. *Numerical analysis*. 6th ed. Brooks/Cole Publishing Company. 1997.

[4] Intel® Math Kernel Library Quick Referente. *Document Number: 253425-011US*. Disponible: http://www.intel.com/software/products/mkl/docs/mklqref/index.htm

[5] L. Snyder. *A Programmer's Guide to ZPL*. The MIT Press, 1999.

[6] Nakamura, Shoichiro. *Análisis numérico y visualización gráfica*. Prentice-Hall Latinoamericana, S. A. México 1996.

[7] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. Journal of Supercomputing, 23(1), 2002.