Caracterización vial en base a nubes de puntos LiDAR terrestre con MPI

A. M. Esmorís, J. C. Cabaleiro, D. L. Vilariño y F. F. Rivera [1]

Resumen— **Este trabajo se centra en la implementación paralela de un algoritmo aplicable a nubes de puntos LiDAR terrestre, que corresponde a la primera etapa de un conjunto de soluciones dedicadas la detección de bordillos. Se proponen distintas alternativas para abordar el problema de una carga de trabajo desbalanceada y se analiza el impacto de la configuración de uso de la infraestructura computacional del CESGA en el rendimiento. Tras analizar los resultados se ha visto que la estrategia de balanceo dinámico de la carga propuesta ha dado mejores resultados que las dos estrategias de balanceo estático probadas. Finalmente, se ha observado que usar una configuración de ejecución basada en reducir la congestión del bus de memoria contribuye a mejorar significativamente el rendimiento.**

Palabras clave— **MPI, LiDAR, Paralelismo, Computación Distribuida, HPC**

I. Introducción

La tecnología LiDAR terrestre [9][10] aplicada a entornos urbanos, específicamente a la segmentación, clasificación y análisis de elementos viales, supone el procesamiento de grandes volúmenes de datos. Concretamente, estos datos se han obtenido con tecnología LiDAR terrestre móvil [11]. Las nubes de puntos han sido generadas mediante un vehículo dotado con escáneres LiDAR [12] que recorre las zonas deseadas. De manera más específica, por cada metro cuadrado pueden tenerse cientos o miles de puntos distribuidos de modo no uniforme, lo que da lugar a etapas de procesamiento que necesitan un tiempo relevante y limitante en muchas aplicaciones para completarse.

En este trabajo se presenta la paralelización [2][3] de una etapa de procesamiento a la que se le ha llamado *Algoritmo de adyacencia*, que supone el coste computacional más elevado de entre todas las operaciones necesarias. Se utiliza MPI [4][5], una librería para el paso de mensajes en entornos de computación de altas prestaciones. Además, se usa el supercomputador FinisTerrae-II del Centro de Supercomputación de Galicia (CESGA) [1] y se analiza el rendimiento con distintas configuraciones.

Para analizar el rendimiento, se ha tomado como referencia una nube de puntos que corresponde a una calle de la ciudad de A Coruña, con aproximadamente 15 millones de puntos. De entre todas las operaciones que se realizan sobre esta nube, la que más tiempo necesita es la aplicación del algoritmo de adyacencia, que supone casi un 50 % del tiempo de ejecución total. Esto puede comprobarse en los datos presentados en la tabla I, en la cual se expone el conjunto de todas las operaciones que realiza el clasificador de elementos viales para generar el resultado final. En dicha tabla se exponen las etapas de procesamiento cuya ejecución tarda más tiempo, agrupando las etapas menos costosas en la categoría de otros procesamientos por un lado y las operaciones relativas a entrada/salida de datos por el otro.

TABLA I
Tabla de tiempos de ejecución secuencial del clasificador vial

OPERACIÓN	TIEMPO (s)
Alg. de adyacencia	685.89
Alg. de contagio	183.80
Obtener línea virtual	379.58
Proporcionalidad por puntos	49.09
Reconstruir franjas	24.53
Otros procesamientos	4.52
Entrada/Salida	74.93
Total	1401.76

II. Algoritmo de adyacencia

El algoritmo de adyacencia recorre de manera iterativa la nube de puntos, procesando aquellos que no estén previamente etiquetados ni como puntos de carretera ni como puntos de señal horizontal. Por cada uno de estos puntos se determinan sus vecinos en un radio determinado y se considera el punto de carretera más cercano como el CRP (*Closest Road Point*) del punto que se está procesando. A continuación, se aplican dos etapas de procesamiento, salvo que para alguna de ellas se haya especificado un valor de entrada nulo:

1. Procesamiento de obstáculo vertical. En aquellos casos en que exista un punto vecino al que se está analizando, cuya altura supere el umbral especificado, no se considerará que el punto que se está procesando corresponda a un bordillo.
2. Procesamiento de diferencia de altura. Cuando la diferencia de altura entre el punto que se está analizando y su CRP supere un máximo indicado como argumento, no se considerará el punto que se está procesando como punto de bordillo.

Es importante tener en cuenta que el hecho de que un punto no se considere como punto de bordillo tras este procesamiento no implica que no vaya a ser clasificado como tal en etapas posteriores.

El comportamiento del algoritmo de adyacencia se describe en el pseudocódigo del algoritmo 1.

[1]Centro Singular de Investigación en Tecnoloxías Intelixentes, CiTIUS, USC

Algorithm 1 Algoritmo de adyacencia

```
for all p in points do
    compute(p)
end for
function compute(p)
    neighs ← points in radius r from p
    ra ← false // ra := road adjacency flag
    aa ← false // aa := above adjacency flag
    for all q in neighs do
        if q is road then
            ra ← true
            if q is closest to p than crp then
                crp ← q
            end if
            if q.z − p.z > th_1 then
                aa ← true
            end if
        end if
    end for
    hdiff ← p.z − crp.z
    if ra and not aa and hdiff > 0 then
        if th_2 is null or hdiff ≤ th_2 then
            p.curb ← true
        end if
    end if
end function
```

III. Balanceo estático de la carga de trabajo

Se ha observado que en la versión paralela el desbalanceo de la carga es muy alto, ya que el algoritmo de adyacencia presenta un lazo cuyo coste depende del tipo y número de vecinos de cada punto y ese valor cambia mucho en las nubes de puntos valoradas. A continuación, se introducen las técnicas utilizadas para aliviar este problema de rendimiento.

A. Distribución por bloques

La primera estrategia consiste en distribuir la carga de trabajo asignando bloques de puntos consecutivos a cada procesador. En primer lugar, se recorre la nube de puntos para determinar cuáles han de ser procesados, estableciendo que no se deben considerar los puntos de carretera ni de señal horizontal como candidatos a bordillo y que, por tanto, no supondrán carga computacional. Una vez conocido el total de puntos que se debe procesar, se dividen en bloques de puntos consecutivos de igual tamaño y se reparten equitativamente entre todos los procesos.

B. Distribución cíclica

Atendiendo a la distribución de las nubes de puntos, puede observarse una tendencia a que los puntos próximos en el archivo de datos estén distribuidos en un conjunto de curvas irregulares, tal y como se ilustra en la figura 1. Generalmente en estas curvas, cuanto más central sea el punto respecto de la misma, mayor será el número de vecinos en un radio reducido y, en consecuencia, mayor será la carga computacional derivada de su procesamiento.

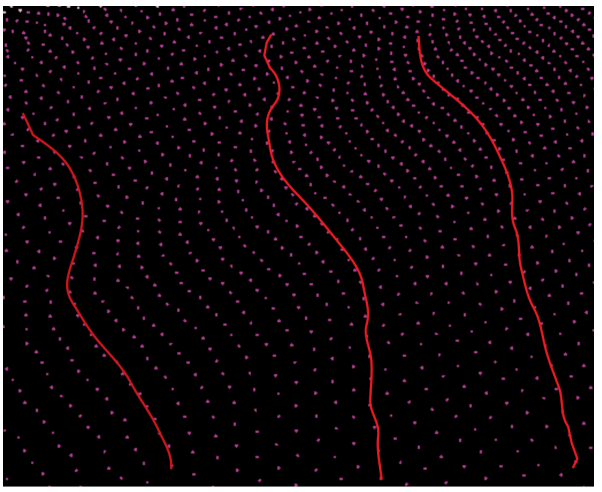

Fig. 1
Distribución de puntos LiDAR terrestre

Esto no siempre se cumple, ya que puntos próximos en el espacio no tienen por qué estar necesariamente próximos en el archivo de datos. Aspectos como pertenecer a distintos barridos, o el procesamiento de los datos originales por un software de terceros para obtener la nube de puntos, explican este hecho.

La falta de relación directa entre la proximidad espacial geográfica y la proximidad en el archivo de datos puede verse más claramente en las nubes de puntos que presentan rotondas. Pudiendo observarse, también, cuando el vehículo que porta el sensor hace un recorrido en el que varía la dirección significativamente, de manera que vuelva a obtener datos de zonas por las que ya había pasado previamente.

Pese a que la distribución de los puntos vecinos expuesta no es siempre uniforme, se considera que es lo suficientemente frecuente como para favorecer distribuciones más balanceadas de la carga de trabajo basándose en ella. Por consiguiente, se propone realizar una distribución basada en un esquema cíclico como segunda alternativa.

IV. Carga de trabajo dinámica temporizada

Incluso realizando una distribución cíclica, que tiende a un reparto más uniforme de la carga de trabajo que la distribución por bloques, ambos métodos dan lugar a una asignación de carga de trabajo poco equilibrada. Este desbalanceo queda patente en la tabla II, donde se muestra para distinto número de procesos el tiempo que tardaron el primer y último proceso en terminar. Dicho desbalanceo redunda directamente en una escasa escalabilidad.

Con el fin de ajustar mejor la distribución de la carga de trabajo, se ha diseñado e implementado un algoritmo de balanceo de carga dinámico que trata de solapar computacion y comunicaciones [6] basándose en el uso de funciones MPI no bloqueantes [7][8]. Dicho algoritmo se divide en dos partes, el sistema de balanceo de carga y el esquema de comunicaciones.

TABLA II

TABLA DE TIEMPO MÁXIMO Y MÍNIMO

N-Procesos	T min	T max	TΔ
4	288.98s	304.38s	15.4s
6	221.8s	256.06s	34.26s
8	193.3s	239.9s	46.6s
12	166.8s	185.9s	19.1s
16	141.3s	173.8s	32.5s
18	128.5s	171.6s	43.1s
24	125.79s	190.8s	65.01s

A. Sistema de balanceo de carga

El sistema de balanceo de carga propuesto se basa en poder monitorizar de manera eficiente, en un instante dado, la carga de trabajo completada o *CWL* (*Completed Workload*) por cada proceso. Esta medida debe estar definida en el intervalo [0,1], como la fracción del trabajo total ya realizado.

Sea W el vector que contiene todos los procesos, se ordena de mayor a menor *CWL* mediante el algoritmo Quicksort [13] y se divide en dos vectores, siendo C el vector que contiene la primera mitad y S el vector que contiene la segunda mitad. En caso de haber un número impar de procesos, siempre será el vector C el que tenga un proceso más que el vector S. De esta manera, el proceso con mayor carga de trabajo completada será el primer elemento en el vector C y el proceso con la menor carga de trabajo completada será el último elemento en el vector S.

Además de la métrica de carga de trabajo completada, el algoritmo debe ser configurado mediante dos parámetros adicionales:

1. WR_{th}. Sirve como umbral de carga de trabajo y, debe haber al menos un proceso que haya completado esta cantidad de carga de trabajo antes de que tenga lugar ninguna redistribución. Ningún proceso que no haya alcanzado este umbral aceptará recibir trabajo de otro proceso. Así, se puede controlar el comienzo de las redistribuciones y favorecer que haya pocas comunicaciones de gran tamaño, que suelen ser preferibles –por el overhead intrínseco a cada comunicación– a muchas comunicaciones de tamaño reducido.
2. WR_{diff}. Sólo se intercambiará trabajo entre pares de procesos cuya diferencia de carga de trabajo supere el valor de este parámetro.

El proceso de redistribución del trabajo (R), asumiendo n procesos y suponiendo que se supera el umbral WR_{th}, es de la siguiente manera:

$$\forall\, i\; c_i \in C\,,\; s_i \in S$$

$$\forall\, 0 \leq i < \left\lceil \frac{n}{2} \right\rceil \; [CWL_i \geq (CWL_{n-i-1} + WR_{diff})]$$

$$\Rightarrow \; R(C_i, S_{\lfloor \frac{n}{2} \rfloor - i - 1})$$

En cuanto a la cantidad de trabajo redistribuido (RWL) desde la perspectiva de S, viene determinado por la siguiente expresión:

$$c \in C\,,\; s \in S$$

$$RWL(c,s) =$$

$$min[0.5, (CWL_c - CWL_s) \cdot (1 - CWL_s)]$$

En el segundo parámetro del mínimo de la anterior formulación, el primer factor corresponde a la diferencia de carga de trabajo entre c y s. Por otro lado, el segundo factor hace alusión a la carga de trabajo pendiente de s. Además, cabe tener en cuenta que existe una restricción técnica impuesta por las comunicaciones, dado que la carga de trabajo nunca podrá superar el tamaño prefijado del buffer de comunicaciones. También, cuando un proceso haya completado toda su carga de trabajo, la proporción de trabajo que reciba de s será siempre de la mitad, es decir, 0.5. Por último, tras la redistribución, las cargas de trabajo del par de procesos implicados quedarán tal que:

$$R_t(c,s) \Rightarrow \begin{cases} CWL_{c_t} < CWL_{c_{t-1}} \\ CWL_{s_t} = CWL_{s_{t-1}} + RWL(c,s) \end{cases}$$

Cabe mencionar que, la carga de trabajo de c tras la redistribución no se puede precisar con exactitud. Esto se debe a que el número de iteraciones computadas por c es desconocido desde la perspectiva de s. Además, la nueva carga no es deducible a partir del *CWL* de c, puesto que éste puede haber participado ya en otros procesos de redistribución. En consecuencia, no es posible conocer, desde la perspectiva de s, la proporción de carga de trabajo que supone para c la cantidad de trabajo redistribuida.

En la figura 2 se ilustra, mediante un ejemplo, el proceso de redistribución de carga de trabajo con una configuración tal que $WR_{diff} = 0.05$, donde cada procesador trabaja a un ritmo diferente (P0 y P4 en intervalos de 0.2, P1 y P3 en intervalos de 0.1, P2 en intervalos de 0.05 y P5 en intervalos de 0.15). De no realizar operaciones de balanceo de carga, se producirá una pérdida importante de rendimiento.

t (instante)	CWL POR PROCESO (P0,	P1,	P2,	P3,	P4,	P5)
t0	0.2	0.1	0.05	0.1	0.2	0.15
t1	0.0575	0.19	0.1925	0.145	0.11	0.105
t2	0.2575	0.29	0.2425	0.245	0.31	0.255
t3	0.2575	0.29	0.29363	0.245	0.25887	0.255
t4	0.4575	0.39	0.34363	0.345	0.45887	0.405
t5	0.38381	0.39	0.41927	0.41869	0.38323	0.405

Fig. 2

EJEMPLO DE DISTRIBUCIÓN DE CARGA DE TRABAJO

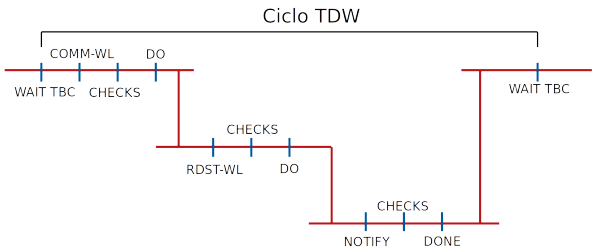

Fig. 3
Ciclo del esquema de comunicaciones

Para ilustrar esto, se establece que los instantes pares (t0, t2, t4) corresponden al avance computacional de cada proceso y los instantes impares (t1, t3, t5) al estado de la carga de trabajo tras las redistribuciones oportunas.

En un primer momento (t_0), la diferencia entre el proceso con mayor CWL (P0) y el proceso con menor CWL (P2) supera el umbral $WR_{diff} = 0.05 \leq 0.15 = CWL_0 - CWL_2$. Por tanto, se produce una redistribución del trabajo, tal que P2 envía parte de su carga a P0. En concreto, envía un 14.25 %, que viene de $min[0.5\ ,\ (CWL_0 - CWL_2) \cdot (1 - CWL_2) = 0.1425]$. De manera análoga, ocurre lo mismo entre P4 y P1, ya que la diferencia de carga de trabajo entre estos supera el umbral, por lo que P1 envía parte de su carga de trabajo a P4. También entre P5 y P3, donde P3 envía parte de su carga de trabajo a P5.

En el instante t_1, se puede ver como queda la carga de trabajo de los procesos tras la redistribución para, en el instante t_2, evolucionar tras otra etapa de computación. A continuación, se produce otra redistribución. En este caso, únicamente la diferencia de carga de trabajo entre P4 y P2 supera el umbral $WR_{diff} = 0.05 \leq 0.0675 = CWL_4 - CWL_2$.

El proceso se repite de la misma manera y, al llegar al instante t_4, la carga de trabajo se distribuye entre P4 y P2 y entre P0 y P3. En caso de seguir desarrollando la evolución, se repetiría la misma mecánica hasta que el trabajo hubiese terminado.

B. Esquema de comunicaciones

En la figura 3 se ilustra el esquema de comunicaciones completo, asumiendo que se produce al menos una redistribución de la carga de trabajo.

En primer lugar, se espera un tiempo *TBC* (*Time Between Communications*). A continuación, los distintos procesos se comunican su carga de trabajo mediante la colectiva no bloqueante *MPI_Iallgather*, al llegar al momento marcado como COMM-WL. Después, se solapan computaciones con comprobaciones del estado de la colectiva no bloqueante (CHECKS) hasta que, finalmente, se comprueba que la comunicación ha terminado. Llegados a este punto, si no fuese a tener lugar ninguna redistribución, el ciclo terminaría y se esperaría un tiempo *TBC* antes de volver a comenzar y comunicar la carga de trabajo. No obstante, en caso de que la configuración de las cargas de trabajo dé lugar al menos a una redistribución, se procedería a realizar la segunda etapa.

En la segunda etapa, los procesos que participan en alguna redistribución llegan al momento marcado como RDST-WL. Aquí, los procesos que van a enviar carga de trabajo comienzan una comunicación punto a punto no bloqueante como emisores, utilizando la función *MPI_Isend*. Por otro lado, los procesos que van a recibir carga de trabajo hacen lo propio como receptores, utilizando la función *MPI_Irecv*. En cuanto a aquellos procesos que no van a participar en ninguna redistribución, saltan directamente a la tercera etapa. Volviendo a los procesos implicados en la redistribución, solapan computación con comprobaciones sobre el estado de la transmisión de carga de trabajo (CHECKS), hasta que comprueban que la comunicación ha terminado y avanzan a la tercera etapa.

Finalmente, al llegar a la tercera etapa, todos los receptores envían una notificación a todos los demás procesos para indicar que han terminado de recibir la carga de trabajo (NOTIFY), utilizando para ello la colectiva no bloqueante *MPI_Ibcast*. Todos los procesos realizan comprobaciones por cada notificación que esperan recibir (CHECKS), hasta que tengan constancia de que han terminado todas las redistribuciones pendientes. Al término de esta etapa, se considera el ciclo concluido y se espera un tiempo *TBC* antes de comenzar el siguiente, aprovechando el periodo de espera para computar sin atender a las comunicaciones.

Cabe mencionar que todos los procesos llevan cuenta en un vector *RIP* (*Redistributions In Process*) de las notificaciones que deben recibir. Esto es necesario porque el vector *RIP* indica en que momento se ha terminado la redistribución, mientras que el *MPI_Request* asociado a la colectiva sirve para comprobar cuando se ha completado la notificación.

En el caso del receptor, tan pronto como termina de recibir la carga de trabajo actualiza el valor en la posición correspondiente del vector *RIP*, sin que los otros procesos dispongan del valor actualizado hasta que termine la comunicación. Sin embargo, el ciclo no debe terminar hasta que todos los procesos tengan el vector *RIP* actualizado, lo cual sólo se puede establecer con seguridad cuando se cumplen las dos condiciones que siguen:

1. Todos los valores de *RIP* están a 0.
2. Todas las *MPI_Request* asociadas a notificaciones han terminado, es decir, han sido consumidas por la función *MPI_Test*.

En la figura 4, el instante t_0 corresponde al momento en que P0 termina de recibir parte de la carga de trabajo de P1 y, por consiguiente, actualiza su vector *RIP* oportunamente. No obstante, P1, P2 y P3 todavía no han recibido la notificación y no tienen su vector *RIP* actualizado. Si no se requiriese la doble comprobación para terminar el ciclo, P0 consideraría que ha concluido y comenzaría la espera del siguiente ciclo antes de que los otros procesos hu-

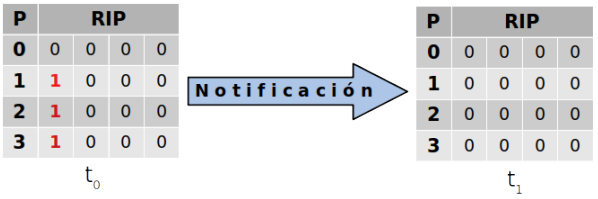

Fig. 4
Tabla RIP

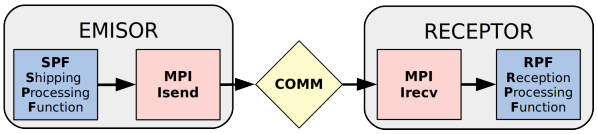

Fig. 5
Gestión del envío y recepción de la carga de trabajo

biesen terminado el ciclo anterior. Por tanto, todas las comunicaciones no bloqueantes se sincronizan en cierta manera al retrasar el comienzo del siguiente ciclo hasta que todos los procesos hayan terminado el anterior. Como puede verse en el instante t_1, que tiene lugar una vez ha concluido la notificación, todos los procesos han constatado que se ha terminado el ciclo y pueden comenzar con la espera que marca el comienzo del siguiente.

C. Caso de uso

El algoritmo de carga de trabajo dinámica temporizada se ha implementado de tal manera que, para aplicarse a un código MPI que realiza computaciones en un bucle, baste con cumplir el pseudocódigo del algoritmo 2.

Algorithm 2 Algoritmo de carga de trabajo dinámica temporizada

```
conf ← adequate settings
TDW_start(conf)
for i = 0 to n do
  if conf.modulo ≡ 0 mod n then
    TDW_balance(CWL)
  end if
  compute(i)
  while i=n-1 and TDW_todo() do
    TDW_balance(CWL)
  end while
end for
```

El primer paso consiste en establecer los parámetros de configuración. En el segundo paso, la configuración es utilizada para iniciar el algoritmo, justo antes de comenzar el bucle distribuido (*TDW_start*). Al comienzo del bucle, se comprueba si la iteración es congruente con cero en el módulo especificado y, en caso de serlo, se invoca a la función *TDW_balance*. Dicha función se encarga de comprobar si ha transcurrido el tiempo necesario y, en caso afirmativo, inicia el proceso de redistribución, tal y como se ha explicado en las subsecciones IV-A y IV-B. En la parte final del bucle se ejecuta otro lazo anidado que captura el comportamiento tras procesar la última iteración, mientras quede trabajo pendiente. Este bucle es necesario para garantizar que todos los procesos terminen y evitar que se den escenarios de espera indefinida. Tal situación podría producirse si algún proceso continuase su flujo de ejecución tras haber terminado su trabajo, mientras los demás esperan indefinidamente por su mensaje para completar sus comunicaciones.

En cuanto a la configuración, además de los parámetros ya explicados, permite especificar dos funciones que se ilustran en la figura 5:

- *SPF* (*Shipping Processing Function*). Esta función se encarga de realizar los procesamientos oportunos para preparar el envío. Su invocación tiene lugar justo antes de la función *MPI_Isend* y se vale de 6 parámetros:
  1. Buffer de envío. Contiene los datos relativos a la carga de trabajo que se delega.
  2. Tamaño. Máxima cantidad de bytes que soporta el buffer de envío.
  3. Destino. Rango del proceso receptor.
  4. *CWL* del emisor. Carga de trabajo completada del proceso que envía la carga de trabajo.
  5. *CWL* del receptor. Carga de trabajo completada del proceso que recibirá la carga de trabajo.
  6. Argumentos extra. Este parámetro se deja a disposición del usuario, para que pase a la función todo aquello que considere oportuno para procesar el envío.
- *RPF* (*Reception Processing Function*). Esta función se encarga de aplicar los procesamientos oportunos a los datos recibidos. Su invocación tiene lugar justo después de la terminación de la recepción asociada a la función *MPI_Irecv* y se vale de 3 parámetros:
  1. Buffer de recepción. Contiene los datos relativos a la carga de trabajo que se asume.
  2. Tamaño. Máxima cantidad de bytes que soporta el buffer de recepción.
  3. Argumentos extra. Este parámetro se deja a disposición del usuario, para que pase a la función todo aquello que considere necesario para procesar la recepción.

V. Análisis de rendimiento

Se ha realizado una comparativa entre las distintas técnicas aplicadas, ejecutando el software con un número de procesos desde 1 hasta 16, obteniendo el tiempo de ejecución en cada caso.

Como puede observarse en la figura 6, la técnica de balanceo de carga dinámico (TDW) presenta menor tiempo de ejecución y, por tanto, un mayor speed-up que las otras dos técnicas analizadas. En cuanto a la distribución estática de la carga de trabajo, resulta evidente que la distribución cíclica (RR) ofrece

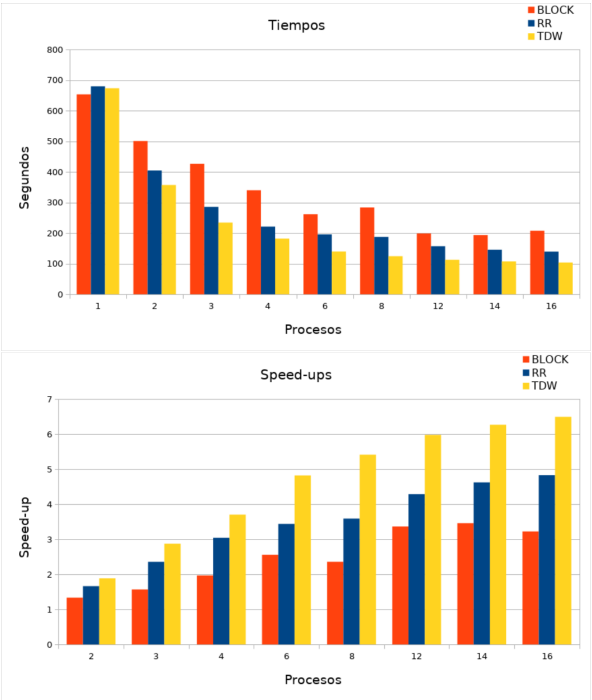

Fig. 6
Rendimiento en 1 nodo

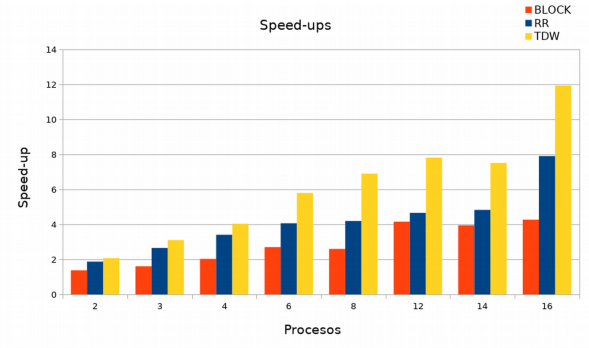

Fig. 7
Rendimiento con 4 procesos por nodo

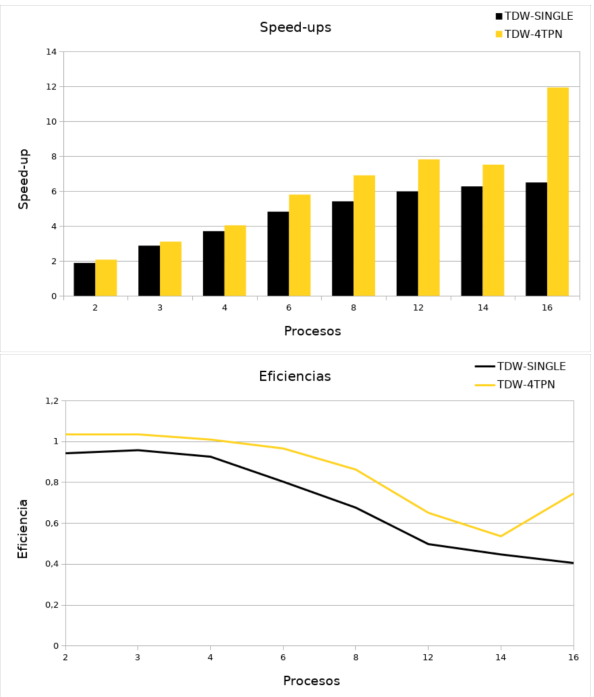

Fig. 8
Rendimiento para 4 procesos por nodo y para un solo nodo

mejores resultados que la distribución por bloques (BLOCK).

Si se atiende al comportamiento de la escalabilidad se puede observar que, aun en el mejor de los casos con 16 procesos, el tiempo más reducido es de 103.71 segundos, que supone un speed-up de 6.49. A raíz de esto, se decide estudiar distintas configuraciones, pues se sospecha que puede estar ocurriendo un problema de congestión del bus de memoria [15]. Para ello, se vuelven a realizar las ejecuciones configurando un límite de 4 procesos por nodo. De esta manera, al utilizar 16 procesos no se ejecutarán todos en el mismo nodo, sino que habrá 4 procesos en 4 nodos. Los resultados correspondientes a esta configuración pueden verse en la figura 7 y suponen un aumento general del rendimiento para todas las estrategias. En términos de speed-up, esto implica que el mejor valor que se obtenía (6.49) al ejecutar 16 procesos en un solo nodo suba hasta alcanzar una cota de 11.94 al ejecutar 16 procesos con un máximo de 4 procesos por nodo.

Como se puede ver en las gráficas de la figura 8, la eficiencia de la técnica varía según la configuración de la infraestructura computacional. La hipótesis de que estaba ocurriendo un problema de congestión del bus de memoria parece confirmarse, ya que encaja con la evolución a un mejor rendimiento que se obtiene al distribuir los procesos entre varios nodos. Otra posible solución a este problema son los algoritmos conscientes de la congestión [16], que tratan de encontrar un equilibrio entre la optimización de la localidad de los datos y la congestión del bus de memoria.

Retomando el análisis de rendimiento se observa que, limitando a un máximo de 4 procesos por bus de memoria, se consiguen mejores resultados. Esto es bastante significativo, ya que se asume un sobrecoste derivado de las comunicaciones, que ya no ocurren dentro del mismo nodo, sino que deben realizarse a través de la red InfiniBand[14] del FinisTerrae-II.

Por último, tanto para terminar de contrastar la hipótesis, como para poner a prueba la escalabilidad de la mejor solución, se han llevado a cabo ejecuciones hasta un máximo de 64 procesos. Como puede verse en las gráficas de la figura 9, la configuración de 2 procesos por nodo alcanza su mejor rendimiento con 36 procesos. A partir de este número de procesos no se consigue un mejor rendimiento, ni siquiera a costa de la eficiencia.

En cuanto al caso de 4 procesos por nodo, su rendimiento a partir de los 16 procesos es peor que el

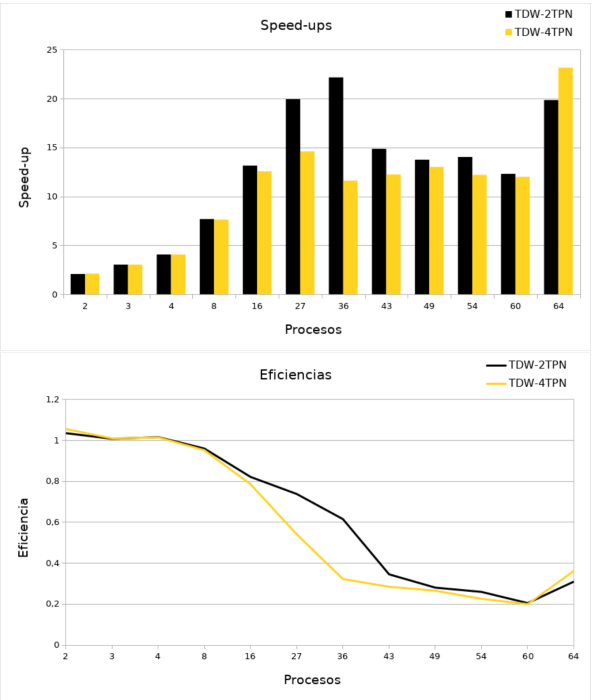

Fig. 9
Rendimiento para 4 y 2 procesos por nodo

obtenido con la configuración de 2 procesos por nodo. No obstante, al llegar a los 64 procesos tarda 28.88 segundos, frente al mejor tiempo de la configuración de 2 procesos por nodo (30.32 segundos). Sin embargo, la eficiencia es peor en comparación, a causa de que esa ligera mejora en el tiempo requiere de un número más elevado de procesos. Así pues, se gastan muchos más recursos computacionales (64 entidades computacionales, en este caso 64 cores en 16 procesadores) para obtener un resultado prácticamente equivalente al que se consigue con una configuración de 2 procesos por nodo (36 entidades computacionales, distribuidas en 36 cores de 18 procesadores).

VI. Conclusiones

Se ha desarrollado una aplicación paralela para la caracterización de elementos viales con LiDAR terrestre en la que la rutina que necesita mayor tiempo de ejecución puede mejorar su rendimiento significativamente a través de un balanceo de la carga computacionalmente eficiente. En este trabajo se presenta una estrategia que mejora el rendimiento para este tipo de problemas.

En cuanto a los métodos de paralelización propuestos, la técnica de balanceo dinámico de la carga de trabajo ha conseguido el menor tiempo de ejecución, el mayor speed-up y la mejor eficiencia. Por tanto, puede concluirse que ha sido el mejor método de paralelización de entre todos los estudiados.

Se ha probado que adaptar el programa a las características computacionales de su contexto de ejecución puede afectar más que significativamente al rendimiento. Concretamente, se obtiene un mejor rendimiento al limitar el número de procesos por nodo, lo que se explica por la reducción de la congestión del bus de memoria en los procesadores.

Por último, se considera de interés, como vía de trabajo futuro, utilizar las señales del sistema operativo [17] para mejorar el algoritmo de balanceo dinámico. Proponiendo sustituir la espera activa por una señal que notifique al proceso cuando tenga que realizar la comprobación oportuna.

Agradecimientos

Este trabajo fue financiado en parte por Babcock International Group PLC (Civil UAVs Initiative de la Xunta de Galicia), la Dirección General de Tráfico (Proyecto BIG-GEOMOVE, SPIP2017-02340), el Ministerio de Educación, Cultura y Deporte, [Proyecto TIN2016-76373-P], la Xunta de Galicia [Proyectos GRC R2016/045 y R2016/037], la Consellería de Cultura, Educación e Ordenación Universitaria (acreditación 2016-2019, ED431G/08, ED431C 2018/2019) y la Unión Europea (European Regional Development Fund - ERDF).

Así mismo, el análisis de rendimiento de las distintas técnicas y configuraciones fueron posibles gracias al Centro de Supercomputación de Galicia (CESGA), que ofreció su infraestructura computacional para la realización de este trabajo.

Referencias

[1] Centro de Supercomputación de Galicia (CESGA), `http://cesga.es`

[2] Thomas Rauber, Gudula Rünger, *Parallel Programming for Multicore and Cluster Systems* Springer, Berlin, Heidelberg, 2013

[3] Peter S. Pacheco, *Parallel Programming with MPI* Morgan Kaufmann Publishers, Inc., 1997

[4] Message Passing Interface Forum *MPI: A Message-Passing Interface Standard* , June 4, 2015

[5] William Gropp, Torsten Hoefler, Rajeev Thakur, Ewing Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface* The MIT Press; 1 edition (November 7, 2014)

[6] Antonio Lain, Prithviraj Banerjee, *Techniques to overlap computation and communication in irregular iterative applications* ICS '94 Proceedings of the 8th international conference on Supercomputing, pp. 236-245, 1994

[7] Torsten Hoefler, Andrew Lumsdaine, Wolfgang Rehm, *Implementation and performance analysis of non-blocking collective operations for MPI* SC '07 Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 52, 2007

[8] Taher Saif, Manish Parashar *Understanding the Behavior and Performance of Non-blocking Communications in MPI* Euro-Par 2004 Parallel Processing, pp 173-182, 2004

[9] NOAA. *What is LIDAR?* website, `https://oceanservice.noaa.gov/facts/lidar.html`, 06/25/18.

[10] J. Shan, C.K. Toth, *Topographic laser ranging and scanning: principles and processing* CRC press, 2008

[11] Haala, Norbert and Peter, Michael and Kremer, Jens and Hunter, Graham, *Mobile LIDAR Mapping for 3D Point Cloud Collection in Urban Areas - A Performance Test* In: Proceedings of XXI ISPRS Congress, Beijing, China, July 3-11, 2008

[12] Brent S. SCHWARZ, James A. HASLIM, Nicholas M. ITURRARAN, Michael D. KARASOFF *LiDAR scanner* Uber Technologies Inc, US9470520B2, 2013

[13] C. A. R. Hoare, *Algorithm 63, Partition; Algorithm 64, Quicksort* Communications of the ACM, Vol. 4, p. 321, 1961.

[14] Pfister, G. F, *An Introduction to the InfiniBand^{TM} Architecture. High Performance Mass Storage and Parallel I/O* (pp. 617-632) Austin, Texas: IBM Enterprise Server Group.

[15] John L. Hennessy, David A. Patterson, *Computer Architecture: A Quantitative Approach* Morgan Kaufmann Publishers, Inc., pp. 238-253, 2007

[16] Mulya Agung, Muhammad Alfian Amrizal, Kazuhiko Komatsu, Ryusuke Egawa and Hiroyuki Takizawa, *A Memory Congestion-aware MPI Process Placement for Modern NUMA Systems* IEEE 24th International Conference on High Performance Computing (HiPC), 2017

[17] Michael Kerrisk, *signal(7)* Linux Programmer's Manual 3.22, The Linux Kernel Archives, 2009