Una Biblioteca Numérica Paralela para UPC

Jorge González-Domínguez[1], María J. Martín[1], Guillermo L. Taboada[1], Juan Touriño[1], Ramón Doallo[1] y Andrés Gómez[2]

Resumen— **Unified Parallel C (UPC) es un lenguaje Partitioned Global Address Space (PGAS) con un alto rendimiento y portabilidad en un amplio rango de arquitecturas, tanto de memoria compartida como distribuida. El presente artículo describe una propuesta para una biblioteca numérica paralela implementada en UPC sobre la biblioteca BLAS secuencial. La biblioteca desarrollada aprovecha las particularidades del paradigma PGAS, teniendo en cuenta la localidad de los datos para garantizar un buen rendimiento. La biblioteca ha sido validada experimentalmente, demostrando su escalabilidad y eficiencia.**

Palabras clave— **Computación Paralela, Partitioned Global Address Space (PGAS), Unified Parallel C (UPC), Basic Linear Algebra Subroutines (BLAS).**

I. Introducción

El modelo de programación Partitioned Global Address Space (PGAS) proporciona importantes ventajas sobre modelos más tradicionales. Al igual que en memoria compartida, en PGAS todos los threads comparten un espacio de direccionamiento global. Sin embargo, este espacio está dividido entre los threads, como en el modelo de memoria distribuida. De este modo, el programador puede explotar la localidad de los datos para incrementar el rendimiento y, al mismo tiempo, disfrutar de las facilidades de programación de la memoria compartida. Los lenguajes PGAS constituyen una solución de compromiso entre facilidad de uso y buen rendimiento gracias a la explotacion de la localidad de los datos. De ahí que hayan sido varios los lenguajes PGAS desarrollados hasta la fecha, como Titanium [1], Co-Array Fortran [2] y Unified Parallel C (UPC) [3].

UPC es una extensión paralela del estándar de C. En [4] El-Ghazawi et al. indicaron, a través de una exhaustiva evaluación de rendimiento, que UPC es una alternativa viable frente a otros paradigmas de programación. Barton et al. [5] demostraron que los códigos UPC pueden obtener un buen rendimiento y una buena escalabilidad con miles de procesadores, siempre y cuando se disponga de un soporte adecuado de compilador y runtime de UPC. Actualmente hay compiladores de UPC tanto comerciales como de código abierto para casi todas las máquinas paralelas.

Sin embargo, la ausencia de bibliotecas en UPC es una barrera para una mayor aceptación de este lenguaje. Las rutinas BLAS (Basic Linear Algebra Subprograms) [6, 7] son funciones que proporcionan bloques estándar para operaciones en álgebra matricial. Son muy usadas en computación científica e ingeniería para obtener altos rendimientos mediante la explotación de la jerarquía de memoria. Las PBLAS (Parallel Basic Linear Algebra Subroutines) [8, 9] son conjuntos de rutinas BLAS paralelas desarrollados para asistir a los programadores que trabajan con sistemas paralelos de memoria distribuida. Sin embargo, los lenguages PGAS carecen de este tipo de bibliotecas. En [10] se presenta una biblioteca numérica paralela para Co-Array Fortran, pero se restringe a la definición de estructuras de datos distribuidas basadas en un objeto abstracto. Esta biblioteca utiliza la sintaxis de Co-Array Fortran, embebida en métodos que permiten comunicaciones entre objetos basándose en la información contenida en ellos.

Este artículo presenta una biblioteca paralela para computación numérica en UPC que mejora la programabilidad y rendimiento de las aplicaciones. Esta biblioteca implementa un subconjunto relevante de las rutinas BLAS explotando las particularidades del paradigma PGAS, teniendo en cuenta la localidad de los datos para garantizar un buen rendimiento. Además, las rutinas llaman internamente a funciones BLAS para llevar a cabo la fracción de computación que ha de realizar cada thread. La biblioteca ha sido validada experimentalmente en el supercomputador Finis Terrae [11].

El resto del artículo se organiza de la siguiente forma. La Sección II describe el diseño de la biblioteca: tipos de funciones (compartidas y privadas), sintaxis y principales características. La Sección III explica las distribuciones de datos usadas para la versión privada de las funciones. La Sección IV muestra los resultados experimentales obtenidos en el supercomputador Finis Terrae. La discusión de las conclusiones se lleva a cabo en la Sección V.

II. Diseño de la biblioteca

Cada función BLAS tiene dos versiones, una *compartida* y una *privada*. En la primera los datos de entrada se encuentran en el espacio de memoria global compartido y, por tanto, ya se encuentran distribuidos de antemano entre los procesadores. En la segunda los datos están en la memoria privada de cada thread. En este caso los datos son distribuidos de forma transparente por la biblioteca. De este modo los usuarios pueden usar la biblioteca independientemente del espacio de memoria donde se almacenen los datos, evitándose redistribuciones tediosas y propensas a fallos. La Tabla I lista todas las rutinas implementadas, que conforman un total de 112 funciones (14x2x4).

[1]Grupo de Arquitectura de Computadores, Departamento de Electrónica y Sistemas, Universidad de A Coruña, e-mail: {jgonzalezd,mariam,taboada,juan,doallo}@udc.es

[2]Centro de Supercomputación de Galicia, Santiago de Compostela, e-mail: agomez@cesga.es

TABLA I
FUNCIONES BLAS IMPLEMENTADAS EN UPC[3].

Nivel	Tnombre	Acción
BLAS1	Tcopy	Copia de un vector
	Tswap	Intercambio de los elementos de dos vectores
	Tscal	Multiplicación de un vector por un escalar
	Taxpy	Actualización de un vector usando los datos de otro: $y = \alpha * x + y$
	Tdot	Producto escalar de dos vectores
	Tnrm2	Norma euclídea
	Tasum	Suma del valor absoluto de todos los elementos de un vector
	iTamax	Encuentra el índice del vector con el valor máximo
	iTamin	Encuentra el índice del vector con el valor mínimo
BLAS2	Tgemv	Producto matriz-vector
	Ttrsv	Resolución de un sistema triangular de ecuaciones lineales
	Tger	Producto tensorial de dos vectores
BLAS3	Tgemm	Producto de dos matrices
	Ttrsm	Resolución de un bloque de sistemas triangulares de ecuaciones lineales

El lenguaje UPC tiene dos bibliotecas estándar: la de funciones colectivas (integrada en la especificación del lenguaje, v1.2 [3]) y la biblioteca de entrada/salida [12]. Las funciones compartidas siguen el estilo de las colectivas, es decir, presentan una cabecera con una sintaxis similar a la de la biblioteca de colectivas, lo que facilita su adopción a los programadores de UPC. En estas rutinas la distribución de los datos es proporcionada por el usuario mediante el uso de arrays compartidos con un determinado `block_size` (tamaño de bloque, número de elementos consecutivos con afinidad con el mismo thread). Por ejemplo, la sintaxis de la rutina `dot` (producto escalar de dos vectores) en UPC en su versión compartida es:

```
int upc_blas_ddot(const int block_size,
const int size, shared const double *x,
shared const double *y, shared double
*dst);
```

[3]Todas las rutinas siguen la siguiente convención de nombrado: `upc_blas_[p]Tnombre`, donde el carácter "p" indica que la función es privada, es decir, los arrays de entrada apuntan a memoria privada; el carácter "T" indica el tipo de dato (c=carácter con signo; s="short" con signo; i=entero con signo; l="long" con signo; f=número en punto flotante; d=entero de doble precisión); y `nombre` es el nombre de la rutina en la biblioteca BLAS secuencial.

siendo `x` e `y` los vectores de entrada y `dst` el puntero a la posición de memoria compartida donde se escribirá el resultado del producto escalar; `block_size` el tamaño de bloque de almacenamiento de los vectores de entrada; `size` la longitud de estos vectores. Esta función trata los punteros `x` e `y` como de tipo `shared [block_size] double[size]`.

En el caso de las rutinas BLAS2 y BLAS3, es necesario un parámetro adicional para indicar la dimensión en la cual los datos son distribuidos: de forma consecutiva bien por filas o por columnas. Por ejemplo, la sintaxis de la rutina UPC para resolver un sistema triangular de ecuaciones lineales es:

```
int upc_blas_dtrsv(const
UPC_PBLAS_TRANSPOSE transpose, const
UPC_PBLAS_UPLO uplo, const UPC_PBLAS_DIAG
diag, const UPC_PBLAS_DIMMDIST dimmDist,
const int block_size, const int n, shared
const double *T, shared double *x);
```

siendo `T` la matriz de entrada y `x` el vector tanto de entrada como de salida; `block_size` el tamaño de bloque de almacenamiento de la matriz; `n` el número de filas y columnas de la matriz; `transpose`, `uplo` y `diag` valores enumerados que indican las características de la matriz de entrada; `dimmDist` el valor enumerado que indica si la matriz de entrada está distribuida por filas o por columnas. El significado del parámetro `block_size` depende de este valor `dimmDist`. Si la matriz está distribuida por filas (`dimmDist==upc_pblas_rowDist`), `block_size` es el número de filas consecutivas con afinidad al mismo thread. En este caso, el puntero `T` se trata como de tipo `shared [block_size*n] double[n*n]`. Por otro lado, si la matriz está distribuida por columnas (`dimmDist==upc_pblas_colDist`), `block_size` es el número de columnas consecutivas con afinidad al mismo thread. En este caso, se trata el puntero `T` como de tipo `shared [block_size] double[n*n]`.

El estilo de la sintaxis de las funciones privadas es similar al de las compartidas, pero ahora se omite el parámetro `block_size` ya que en este caso la distribución de los datos es manejada automáticamente por la biblioteca. Por ejemplo, la sintaxis para la rutina `dot` es:

```
int upc_blas_pddot(const int size, const
int src_thread, const double *x, const
double *y, const int dst_thread, double
*dst);
```

siendo `x` e `y` los vectores de entrada, `dst` el puntero a memoria privada donde se escribirá el resultado del producto escalar y `src_thread` y `dst_thread` los threads donde estarán almacenadas las entradas y salidas respectivamente. El número total de threads se puede conocer durante la ejecución de un programa UPC mediante el identificador `THREADS` [13]. A cada thread se le asigna un identificador entero único dentro del rango 0, 1, ..., `THREADS`-1. Los datos

de entrada se encuentran en el thread `src_thread`, mientras que los de salida se escribirán en el thread `dst_thread`. Si `src_thread==THREADS`, la entrada se encontrará almacenada inicialmente en todos los threads. Si `dst_thread==THREADS` la salida será replicada en todos los threads. Las distribuciones de datos usadas en la versión privada de las rutinas son descritas en la siguiente sección.

Todas las funciones llaman internamente a implementaciones BLAS muy optimizadas para llevar a cabo la computación de cada thread, con el objeto de obtener el mayor rendimiento posible. Así, las rutinas desarrolladas no implementan la operación concreta (el producto escalar, el producto matriz-vector, etc.) sino que distribuyen los datos y sincronizan las llamadas a las funciones BLAS secuenciales.

Con el objeto de incrementar el rendimiento de los programas, en UPC es posible utilizar las siguientes optimizaciones:

- Privatización de punteros: el uso de punteros de C estándar en lugar de punteros de UPC a memoria compartida al acceder a datos en el espacio compartido que son afines al thread que los está procesando. Los punteros a memoria compartida requieren más espacio y son más costosos de derreferenciar. Algunas evaluaciones previas [4] muestran que la sobrecarga debida al uso de punteros a memoria compartida incrementa el tiempo de ejecución alrededor de tres órdenes de magnitud.
- Agregación de accesos a espacios de memoria remotos: mediante el uso de copias en bloque, usando `upc_memget` y `upc_memput`.
- Solapamiento de accesos a posiciones de memoria remota con computación: se puede lograr mediante el uso de barreras "split-phase".

Todas estas optimizaciones ha sido incorporadas a la biblioteca siempre que ha sido posible.

III. Distribuciones de datos para las funciones privadas

Los accesos locales en UPC (accesos a posiciones de la memoria compartida con las cuales el thread tiene afinidad) pueden ser 1 o 2 órdenes de magnitud más rápidos que los accesos remotos. Por ello, todas las versiones privadas de las rutinas implementadas usan distribuciones que minimizan los accesos a memoria remota.

Si `dst_threads==THREADS` el subresultado obtenido por cada thread se replica en todos los demás. Se consideraron dos opciones distintas para llevar a cabo la copia de las partes del resultado:

- Cada thread copia su parte de la solución en varios arrays compartidos, cada uno con afinidad a un thread distinto. Esto provoca $n-1$ accesos remotos por cada thread, es decir, $n \times (n-1)$ mensajes.
- Cada procesador copia su parte de solución en un mismo array compartido con afinidad al thread 0. Después, cada thread copia la solución completa a su memoria privada. En este caso el número de accesos remotos es sólo $2 \times n$, aunque la mitad de ellos transfieren una mayor cantidad de datos que la opción anterior.

Una evaluación preliminar de ambas estrategias en el supercomputador Finis Terrae indicó que la segunda opción era claramente la mejor en términos de rendimiento. Por ello, se utilizó en todas las funciones privadas de la biblioteca.

En cuanto a las distribuciones de datos, los arrays compartidos de UPC sólo pueden ser distribuidos en una única dimensión y de forma bloque, cíclica o bloque-cíclica. No es posible distribuirlos por bloques de tamaño arbitrario o ciclos. En [14] se propone una extensión a UPC que permite al programador crear arrays con bloques en varias dimensiones, pero no forma parte del estándar de UPC.

Debido a la similitud de los algoritmos utilizados en cada nivel BLAS, la distribución escogida ha sido la misma para todas las funciones del mismo nivel.

A. *Distribuciones en rutinas BLAS1*

El nivel BLAS1 consta de las funciones que operan con uno o más vectores densos. Las Figuras 1 y 2 muestran la copia final de los resultados a todos los threads para las distribuciones cíclica y bloque. La distribución bloque presenta una clara ventaja: es posible realizar todas las copias con un número reducido de llamadas a `upc_memget` y `upc_memput`.

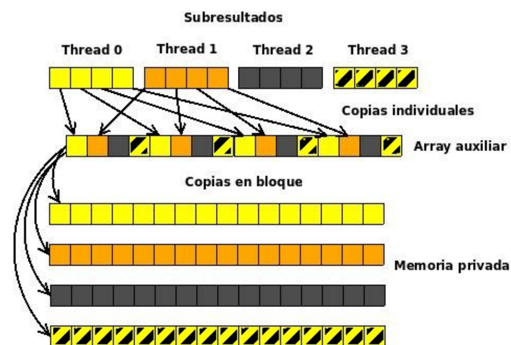

Fig. 1. Movimientos de datos con una distribución cíclica

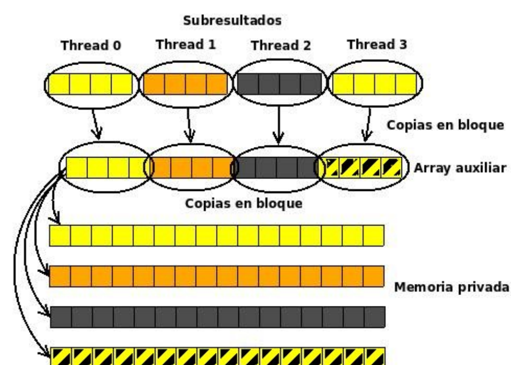

Fig. 2. Movimientos de datos con una distribución bloque

B. *Distribuciones en rutinas BLAS2*

Este nivel contiene operaciones entre una matriz y un vector. Una matriz puede estar distribuida por fi-

las, columnas o bloques. Se usará el producto matriz-vector como ejemplo para explicar la distribución seleccionada.

La Figura 3 muestra el comportamiento de la rutina para la distribución por columnas. Para computar el *i-ésimo* elemento del resultado, se deben sumar todos los valores *i-ésimos* de los subresultados. Cada una de estas sumas globales se realiza mediante una función de reducción de la biblioteca `upc_collective.h`. Otros lenguajes permiten realizar un conjunto de operaciones de reducción de forma eficiente con una única función colectiva. Sin embargo, de momento UPC no dispone de tal función.

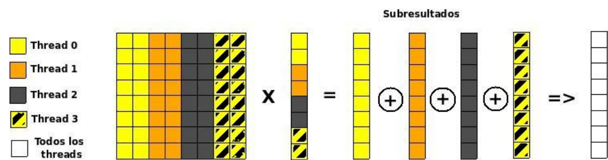

Fig. 3. Producto matriz-vector con distribución por columnas

La Figura 4 muestra el comportamiento para una distribución por filas. Ahora cada thread calcula una parte del resultado final (subresultados en la figura). Estos subresultados sólo se tienen que copiar en la posición que les corresponde.

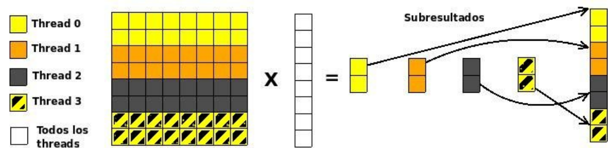

Fig. 4. Producto matriz-vector con distribución por filas

La Figura 5 muestra el comportamiento de la rutina para una distribución de la matriz por bloques (tanto por filas como por columnas). Esta opción lleva consigo también una operación de reducción por cada fila. Además, cada una de estas reducciones debe ser desarrolladas por todos los threads (aunque no todos los threads tengan elementos para todas las filas) ya que las operaciones colectivas de UPC involucran a todos lo threads.

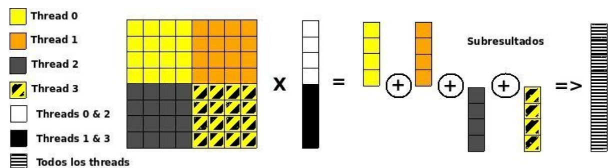

Fig. 5. Producto matriz-vector con distribución en bloques

La evaluación preliminar del rendimiento de estas distribuciones ha confirmado que la distribución por filas es la mejor opción. En [15] Nishtala et al. proponen extensiones para la biblioteca de colectivas de UPC que soporten operaciones sobre subconjuntos. Si finalmente se incluyese esa extensión en la especificación de UPC sería necesario reconsiderar las distribuciones por columnas y bloques.

C. Distribuciones en rutinas BLAS3

Este nivel contiene operaciones entre dos matrices. De nuevo, una matriz se puede distribuir por filas, columnas o de ambas formas (por bloques). Las ventajas e inconvenientes de cada distribución son idénticas a las indicadas en la sección anterior para las funciones BLAS2, seleccionándose por tanto la distribución por filas. La Figura 6 muestra el comportamiento del producto entre dos matrices.

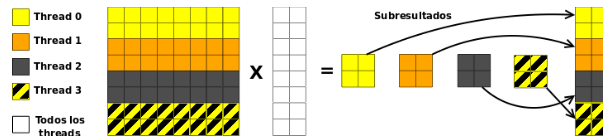

Fig. 6. Producto matriz-matriz con una distribución por filas

En cuanto a la distribución de la matriz, las rutinas que resuelven sistemas triangulares de ecuaciones lineales (`Ttrsv` y `Ttrsm`) conforman un caso especial. En estas rutinas las filas de la matriz no se distribuyen de forma bloque perfecta sino de forma bloque-cíclica. La Figura 7 muestra un ejemplo para dos threads, dos bloques por thread y una matriz de entrada con las siguientes opciones: no transpuesta (`transpose==upc_pblas_noTrans`), triangular inferior (`uplo==upc_pblas_lower`) y con no todos los elementos de la diagonal iguales a 1 (`diag==upc_pblas_nonUnit`).

La matriz triangular está dividida lógicamente en bloques cuadrados T_{ij}. Estos bloques son submatrices triangulares si $i == j$, matrices cuadradas si $i > j$ y matrices nulas si $i < j$.

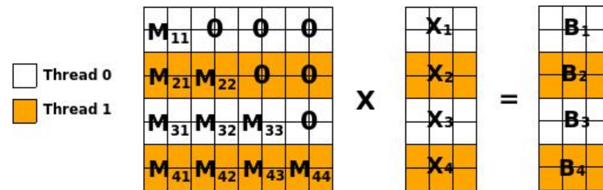

Fig. 7. Distribución de la matriz para `upc_blas_pTtrsm`

El algoritmo usado por esta función se muestra en la Figura 8. Se debe tener en cuenta que todas las operaciones entre dos sincronizaciones son susceptibles de ejecutarse en paralelo. Cuantos más bloques se usen para dividir la matriz, más cálculos se podrán paralelizar pero también se necesitarán más sincronizaciones. Se ha realizado una evaluación experimental para determinar cuál es el mejor número de bloques de acuerdo al tamaño de la matriz y al número de threads.

IV. Resultados experimentales

La evaluación experimental de la biblioteca desarrollada ha sido realizada en el supercomputador Finis Terrae [11] del Centro de Supercomputación de Galicia (CESGA), clasificado en la posición #427 en la lista TOP500 de noviembre de 2008 (14 TFlops). Consta de 142 nodos HP RX7640, cada uno de ellos con 16 núcleos IA64 Itanium2 Montvale a 1.6 GHz,

1 $X_1 \leftarrow$ *Resuelve* $M_{11} * X_1 = B_1$ `Ttrsm()`
2 — *sincronización* —
3 $B_2 \leftarrow B_2 - M_{21} * X_1$ `Tgemm()`
4 $B_3 \leftarrow B_3 - M_{31} * X_1$ `Tgemm()`
5 $B_4 \leftarrow B_4 - M_{41} * X_1$ `Tgemm()`
6 $X_2 \leftarrow$ *Resuelve* $M_{22} * X_2 = B_2$ `Ttrsm()`
7 — *sincronización* —
8 $B_3 \leftarrow B_3 - M_{32} * X_2$ `Tgemm()`
9 ...

Fig. 8. Algoritmo Ttrsm

repartidos en dos celdas (8 cores por celda), con 128 GB de memoria y una tarjeta InfiniBand dual 4X (16 Gbps de ancho de banda teórico efectivo).

La evaluación de rendimiento ha sido realizada con una configuración híbrida (memoria compartida para comunicaciones intra-nodo y red InfiniBand para comunicaciones inter-nodo). Esta arquitectura híbrida es muy interesante en los lenguages PGAS, permitiendo explotar la localidad de los threads que corren en el mismo nodo, así como mejorar la escalabilidad a través del uso de memoria distribuida.

De las posibles configuraciones híbridas, se seleccionó la que hace uso de cuatro threads por nodo, dos por celda. Esta configuración proporciona un buen equilibrio entre el rendimiento de memoria compartida y el acceso escalable a la red InfiniBand. El compilador usado ha sido el Berkeley UPC 2.6.0 [16] y la biblioteca BLAS secuencial la Intel Math Kernel Library (MKL) 9.1 [17].

La Tabla II y la Figura 9 muestran los tiempos de ejecución, eficiencias y speedups obtenidos por la función *pddot* (BLAS1), que realiza el producto escalar de dos vectores de doubles, con los datos de entrada y salida en memoria privada, para diferentes tamaños del vector de entrada y número de threads. Estos tiempos, y los siguientes, han sido obtenidos con las entradas previamente replicadas en todos los threads (src_thread==THREAD) y escribiendo el resultado en el thread 0 (dst_thread==0).

A pesar de que el tiempo total de las computaciones es muy pequeño (del orden de milisegundos), la escalabilidad es relativamente elevada. Sólo el paso de cuatro a ocho threads reduce ligeramente la pendiente de la curva. Esto se debe a que los resultados para ocho threads son los primeros donde se usan comunicaciones sobre InfiniBand, y no sólo memoria compartida.

Los tiempos, eficiencias y speedups del producto matriz-vector (*pdgemv*, un ejemplo de nivel BLAS2) se muestran en la Tabla III y la Figura 10. Se usan matrices cuadradas, con lo que el tamaño representa el número de filas y columnas. De nuevo, los speedups son relativamente elevados a pesar de que los tiempos de ejecución son bastante reducidos.

Por último, la Tabla IV y la Figura 11 muestran los tiempos, eficiencias y speedups de la ejecución de una función BLAS3, el producto de dos matrices (*dgemm*). Las matrices de entrada son también

TABLA II
TIEMPOS(*ms*) DE *pddot* (BLAS1)

PRODUCTO ESCALAR (pddot) (M = millón de elementos por vector)			
Th \ Tam	50M	100M	150M
1	147,59	317,28	496,37
2	90,47	165,77	262,37
4	43,25	87,38	130,34
8	35,58	70,75	87,60
16	18,30	35,70	53,94
32	9,11	17,95	26,80
64	5,22	10,68	15,59
128	3,48	7,00	10,60

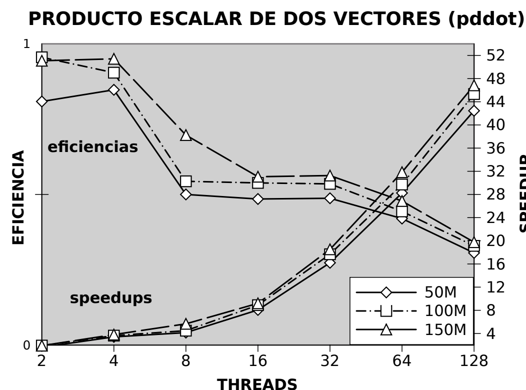

Fig. 9. Eficiencias/speedups de *pddot* (BLAS1)

TABLA III
TIEMPOS(*ms*) DE *pdgemv* (BLAS2)

PRODUCTO MATRIZ-VECTOR (pdgemv)			
Th \ Tam	10.000	20.000	30.000
1	145,08	692,08	1.424,7
2	87,50	379,24	775,11
4	43,82	191,75	387,12
8	27,93	106,30	198,88
16	15,18	53,58	102,09
32	9,38	38,76	79,01
64	4,55	19,99	40,48
128	2,39	10,65	21,23

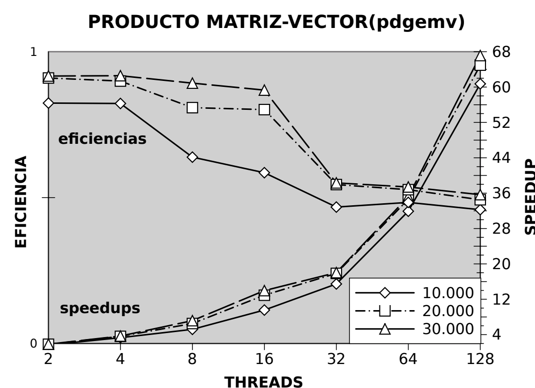

Fig. 10. Eficiencias/speedups de *pdgemv* (BLAS2)

cuadradas. Los speedups son mejores en este caso porque los tiempos de ejecución son mucho mayores (del orden de segundos), permitiendo un mayor aprovechamiento del paralelismo.

Los resultados de las rutinas implementadas en su versión compartida no se muestran porque dependen en gran medida de la distribución escogida por el

TABLA IV
TIEMPOS(s) DE *pdgemm* (BLAS3)

PRODUCTO MATRIZ-MATRIZ (pdgemm)			
Th \ Tam	6.000	8.000	10.000
1	68,88	164,39	319,20
2	34,60	82,52	159,81
4	17,82	41,57	80,82
8	9,02	21,27	41,53
16	4,72	10,99	21,23
32	2,56	5,90	11,23
64	1,45	3,24	6,04
128	0,896	1,92	3,50

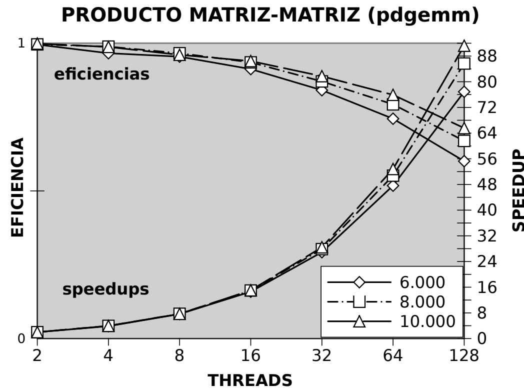

Fig. 11. Eficiencias/speedups de *pdgemm* (BLAS3)

usuario. Si dicha elección es adecuada (la que se usa en las versiones privadas y descrita en la Sección II), se obtienen tiempos y speedups similares.

V. Conclusiones

Según nuestro conocimiento, ésta es la primera biblioteca numérica paralela en UPC. Las bibliotecas numéricas mejoran el rendimiento y programabilidad de las aplicaciones científicas y de ingeniería. Hasta ahora, la utilización de bibliotecas BLAS paralelas estaba restringida a MPI y OpenMP. Gracias a la presente biblioteca, los programadores de UPC pueden hacer uso de estas bibliotecas numéricas, lo que redundará en una mayor adopción del lenguaje.

La biblioteca implementada permite utilizar datos almacenados tanto en memoria privada como compartida. En el primer caso la biblioteca decide la mejor distribución de forma transparente al usuario. En la segunda, la biblioteca trabaja con la distribución porporcionada por él. En ambos casos se han aplicado diversas técnicas de optimización del rendimiento de UPC, como la privatización o el movimiento de datos en bloque.

Las rutinas implementadas hacen uso de una biblioteca BLAS secuencial. Esto no sólo mejora la eficiencia (las bibliotecas BLAS se han estado implementando y optimizando durante más de dos décadas y, por tanto, presentan una extraordinaria madurez en términos de estabilidad y eficiencia) sino que también permite recurrir a nuevas versiones BLAS tan pronto como se encuentren disponibles.

La biblioteca propuesta ha sido evaluada experimentalmente en un supercomputador basado en una arquitectura Intel Itanium y con una red de interconexión InfiniBand, obteniendo una elevada escalabilidad y eficiencia.

En la actualidad nos encontramos extendiendo esta biblioteca para que soporte computación con matrices y vectores dispersos.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia dentro del proyecto (TIN2007-67537-C03-02). Agradecemos al CESGA (Centro de Supercomputación de Galicia) el habernos proporcionado acceso al supercomputador Finis Terrae.

Referencias

[1] "Titanium Project Home Page", `http://titanium.cs.berkeley.edu/`.
[2] "Co-Array Fortran", `http://www.co-array.org/`.
[3] UPC Consortium, *UPC Language Specifications, v1.2*, 2005, `http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`.
[4] T. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: a NPB Experimental Study", in *Proc. 14th ACM/IEEE Conf. on Supercomputing (SC'02)*, Baltimore, MD, USA, 2002, pp. 1–26.
[5] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje and J. N. Amaral, "Shared Memory Programming for Large Scale Machines", in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06)*, Ottawa, Ontario, Canada, 2006, pp. 108–117.
[6] "BLAS Home Page", `http://www.netlib.org/blas/`.
[7] J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.
[8] "PBLAS Home Page", `http://www.netlib.org/scalapack/pblasqref.html`.
[9] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker and R. Clinton Whaley, "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms", in *Proc. 2nd International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science (PARA'95)*, Lyngby, Denmark, 1995, vol. 1041 of *Lecture Notes in Computer Science*, pp. 107–114.
[10] R. W. Numrich, "A Parallel Numerical Library for Co-Array Fortran", in *Proc. Workshop on Language-Based Parallel Programming Models (WLPP'05)*, Poznan, Poland, 2005, vol. 3911 of *Lecture Notes in Computer Science*, pp. 960–969.
[11] "Finis Terrae Supercomputer", `http://www.top500.org/system/9500`.
[12] T. El-Ghazawi, F. Cantonnet, P. Saha, R. Thakur, R. Ross and D. Bonachea, *UPC-IO: A Parallel I/O API for UPC v1.0*, 2004, `http://upc.gwu.edu/docs/UPC-IOv1.0.pdf`.
[13] T. Sterling T. El-Ghazawi, W. Carlson and K. Yelick, *UPC: Distributed Shared Memory Programming*, Wiley-Interscience, 2005.
[14] C. Barton, C. Casçaval, G. Almási, R. Garg, J. N. Amaral, M. Farreras, "Multidimensional Blocking in UPC", in *Proc. 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, Urbana, IL, USA, 2007, vol. 5234 of *Lecture Notes in Computer Science*, pp. 47–62.
[15] R. Nishtala, G. Almási and C. Casçaval, "Performance without Pain = Productivity: Data Layout and Collective Communication in UPC", in *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City, UT, USA, 2008, pp. 99–110.
[16] "Berkeley UPC Project", `http://upc.lbl.gov`.
[17] "Intel Math Kernel Library", `http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm`.