Biblioteca de Operaciones Colectivas para el Lenguaje de Programación Paralela UPC

José Manuel Andión, Guillermo L. Taboada, Juan Touriño y Ramón Doallo[1]

Resumen— **Este artículo presenta una biblioteca de operaciones colectivas para Unified Parallel C (UPC), un lenguaje Partitioned Global Address Space (PGAS) que permite programar una arquitectura de memoria distribuida como si fuese un sistema de memoria compartida. La biblioteca desarrollada: (1) extiende la biblioteca estándar de operaciones colectivas de UPC; y (2) incrementa el rendimiento de las primitivas estándar al aprovechar las particularidades del paradigma PGAS, que permite explotar de forma más eficiente la localidad de los datos. El análisis del impacto en la programabilidad de las nuevas operaciones colectivas propuestas, así como la mejora del rendimiento, evaluada experimentalmente en un clúster multi-core, han permitido aseverar la validez y eficiencia de la biblioteca desarrollada.**

Palabras clave— **Programación Paralela, Operaciones Colectivas, Unified Parallel C (UPC), Partitioned Global Address Space (PGAS), Clúster Multi-core.**

I. Introducción

UPC (Unified Parallel C) es un lenguaje de programación paralela que surge como una extensión a la especificación estándar del lenguaje C. Sigue el modelo de programación PGAS (Partitioned Global Address Space) en el que los distintos threads operan en un espacio de memoria compartido que se encuentra lógicamente particionado. Así, es posible incrementar la programabilidad, a la vez que explotar la localidad de los datos, incrementando de este modo la productividad frente a otras alternativas tradicionales como los paradigmas de paso de mensajes o de memoria compartida. Existen varios compiladores y entornos de ejecución para UPC, tanto Open Source (Berkeley UPC –BUPC–, GCC-UPC o MuPC) como comerciales (HP UPC, Cray UPC o IBM XL Alpha Edition UPC).

En programación paralela es necesario que los distintos threads/procesos trabajen sobre el conjunto de datos del problema de forma coordinada. En este caso las operaciones que simultáneamente involucran a un grupo de threads/procesos se denominan operaciones colectivas. UPC dispone de una biblioteca de operaciones colectivas que forma parte de su especificación. No obstante, su funcionalidad es reducida y son numerosas las propuestas para su extensión. Además, sus implementaciones no suelen ser eficientes.

Así, en el presente artículo presentamos la optimización de algunas de las operaciones colectivas estándar mediante el uso de un algoritmo más eficiente sobre arquitecturas clúster multi-core, a la vez que se permite la selección manual del algoritmo a utilizar. También se ha realizado el diseño e implementación de nuevas operaciones buscando una mayor programabilidad y productividad, proporcionando funcionalidad ampliamente extendida en otros paradigmas como el de paso de mensajes. Por último, se ha realizado un análisis del impacto en términos de programabilidad y productividad así como de rendimiento del lenguaje UPC y la biblioteca desarrollada.

[1]Departamento de Electrónica y Sistemas, Universidade da Coruña, Campus de Elviña, s/n, 15071 A Coruña. E-mail: {jandion,taboada,juan,doallo}@udc.es.

II. Operaciones Colectivas en UPC

La necesidad de comunicación entre los distintos threads en un programa UPC ha llevado a que se incluyese en el estándar del lenguaje [1] una especificación [2] de las operaciones colectivas más comúnmente utilizadas [3]. Estas operaciones colectivas pueden ser: (1) de relocalización de datos, como broadcast, scatter, gather, allgather, intercambio global y permutación; o (2) de computación o consolidación de datos, que realizan algún tipo de cálculo a mayores del movimiento de datos, como son la reducción y el prefijo paralelo.

La motivación de la existencia de operaciones colectivas en UPC, un lenguaje cuyo modelo de memoria compartida podría hacerlas en cierto modo prescindibles, es el aumento del rendimiento y la programabilidad. Así, la transferencia de datos en bloque suele ser más eficiente que los accesos elemento a elemento a la memoria compartida. En efecto, los desarrolladores de bibliotecas pueden proporcionar implementaciones a bajo nivel altamente eficientes que puedan aprovechar la arquitectura del sistema subyacente. Por último, en otros modelos de programación paralela como el paso de mensajes, los programadores están familiarizados con las operaciones colectivas. Al proporcionar UPC esta funcionalidad se está facilitando su adopción.

La Figura 1 representa el funcionamiento de la operación de broadcast en UPC, en donde el bloque de tamaño *nbytes* apuntado por *src* y ubicado en el thread th_0 es transferido al array *dst*. En la Figura 2 se muestra el funcionamiento del allgather.

Son numerosas las extensiones que han sido propuestas para aumentar la funcionalidad de la biblioteca estándar de operaciones colectivas de UPC. Así, en UPC es obligatorio que todos los threads participen en las operaciones colectivas, mientras que en otras soluciones paralelas como MPI es posible que un subconjunto de todos los procesos invoquen una primitiva colectiva. Se ha propuesto evitar esta restricción en UPC a través de la definición de conjuntos de threads (teams) [4], incluso incorporando la semántica de grupos, similar a la existente en MPI [5]. Otra propuesta en este sentido sería la de las

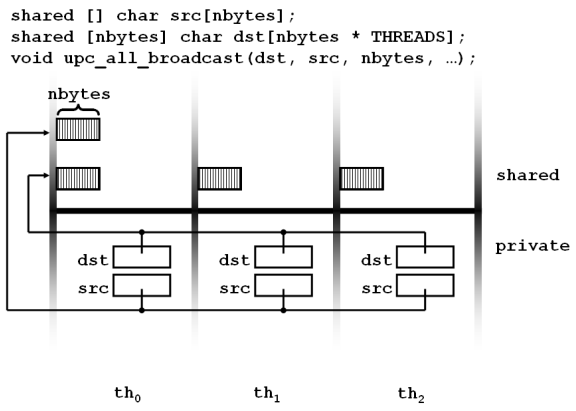

Fig. 1. Ejemplo de `upc_all_broadcast`

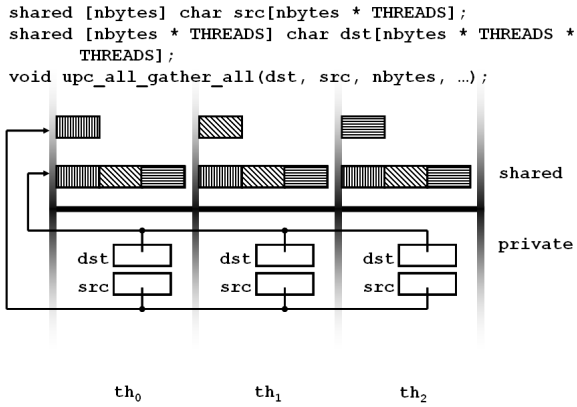

Fig. 2. Ejemplo de `upc_all_gather_all`

colectivas one-sided [6], que serían llamadas por un único thread, pero su funcionamiento afectaría a varios threads, tal como ocurre con las operaciones de memoria one-sided `upc_global_alloc` y `upc_free`.

Las operaciones colectivas estándar de UPC obligan al uso de bloques de datos del mismo tamaño, además de su almacenamiento de forma consecutiva. Con el objetivo de salvar estas restricciones se han definido las collectivas "variable-sized data blocks" [7], que permiten que cada bloque de datos tenga un tamaño distinto, así como operar sobre bloques de datos no contiguos.

Otra propuesta son las colectivas "in-place" [7], que evitan la obligatoriedad de usar dos regiones de memoria disjuntas para los datos origen y destino. Al permitir la sobreescritura de los datos iniciales es posible incrementar la eficiencia de la operación.

Finalmente, existe el propósito de dotar a UPC de colectivas asíncronas [7], evitando la sincronización intrínseca existente. De este modo, sería posible solapar computación local en cada thread con las operaciones de comunicación. Este soporte podría basarse en las operaciones asíncronas de copia de memoria de UPC [8], las cuales permiten transferencias no bloqueantes de datos no contiguos. Conviene destacar que la gran variedad de propuestas en este ámbito es muestra del importante interés y esfuerzo investigador que ha suscitado el desarrollo de nuevas operaciones colectivas en UPC.

III. Nueva Biblioteca de Operaciones Colectivas en UPC

La implementación de una biblioteca de operaciones colectivas en UPC requiere de transferencias de datos entre threads, que es posible realizar mediante asignaciones directas o copias de memoria. La asignación de variables compartidas es la expresión más sencilla y evita la codificación explícita de envíos y recepciones de datos. No obstante, sólo están disponibles para tipos de datos básicos. Las copias/transferencias de memoria son más genéricas, evitando las restricciones de la asignación directa. La implementación de las operaciones colectivas puede realizarse mediante una biblioteca de comunicación a bajo nivel, como GASNet [9] en el caso de Berkeley UPC (BUPC) [10], o mediante operaciones estándar de UPC como `upc_memcpy`, tal como ocurre en la implementación de referencia de las colectivas [11]. Esta implementación de referencia se basa en dos técnicas, *pull* y *push*, diferenciándose en la parte activa de las comunicaciones con `upc_memcpy` (qué thread realiza la llamada). En pull cada thread destino copia los datos que le corresponden del thread origen en paralelo, mientras que en push cada thread origen copia sus datos a todos los threads destino. La selección de una técnica pull o push para la implementación de colectivas debe proporcionar una distribución equitativa de la carga de las comunicaciones. Por ejemplo, en broadcast y scatter será conveniente utilizar pull, donde los threads destino copian los datos en paralelo del thread origen, mientras que la implementación push maximiza el paralelismo de la colectiva gather.

La biblioteca que presentamos en este artículo, presenta las siguientes características: (1) está basada en `upc_memcpy`, con el objetivo de ser portable al estar basada en una operación del estándar de UPC; (2) busca proporcionar varios algoritmos por operación colectiva, seleccionables en tiempo de compilación; (3) implementa extensiones de colectivas ya propuestas, antes que definir nuevas soluciones; y (4) propone nuevas colectivas cuando no existen soluciones previas que recojan esa operación.

A. Selección de Algoritmos en Colectivas UPC

Todas las funciones colectivas estándar de UPC tienen un parámetro de tipo `upc_flag_t` que se utiliza para controlar la semántica de sincronización de datos. En la biblioteca que hemos desarrollado este parámetro también se empleará para permitir la selección de algoritmo. Así, inicialmente se han definido los flags `UPC_PULL`, `UPC_PUSH` y `UPC_MULTICORE`, para los algoritmos de tipo *pull* (de la implementación de referencia), de tipo *push* (de la implementación de referencia), y el algoritmo Multi-core Aware desarrollado en este proyecto.

B. Algoritmo Multi-core Aware

La biblioteca desarrollada incorpora nuestra propuesta de algoritmo "Multi-core Aware" para el incremento de la eficiencia en las operaciones colectivas de UPC sobre arquitecturas clúster multi-core. Este

algoritmo se basa en designar a un thread como *root* en cada nodo, el cual se encargará de las comunicaciones entre nodos. Para asignar este rol de root de un nodo es preciso una llamada a una función de inicialización, común a todas las operaciones colectivas, y que sólo necesita ser invocada en una ocasión por cada thread.

Esta inicialización se basa en la función `long gethostid(void)`, que identifica de forma unívoca y portable (POSIX) a un sistema. Una vez que cada thread obtiene el identificador de su sistema, el thread 0 es el encargado de establecer la estructura de árbol que se extenderá por todo el sistema clúster multi-core, indicando cuales son los threads "root" de cada nodo, y el número de threads que dependen de ellos.

En el presente trabajo se han implementado parte de las funciones de movimiento de datos (broadcast, scatter, gather y allgather) utilizando el algoritmo Multi-core Aware. Estas funciones mantienen la interfaz estándar, a la vez que mediante el flag de sincronización permiten la selección del algoritmo a ejecutar. El funcionamiento de una operación representativa, el broadcast (`upc_all_broadcast`), servirá para ilustrar el funcionamiento de este algoritmo, que trabaja en dos pasos. En el primero de ellos, los threads que han sido designados como roots en su nodo solicitan al thread root de la operación el dato en cuestión. Una vez que lo han obtenido lo enviarán a los restantes threads que dependen de ellos.

La Figura 3 representa el funcionamiento de esta función para el caso en el que th_0 y th_1 están en un nodo y th_2 en otro. En la Figura 4 se muestra el funcionamiento de este algoritmo para la operación de gather en el mismo escenario.

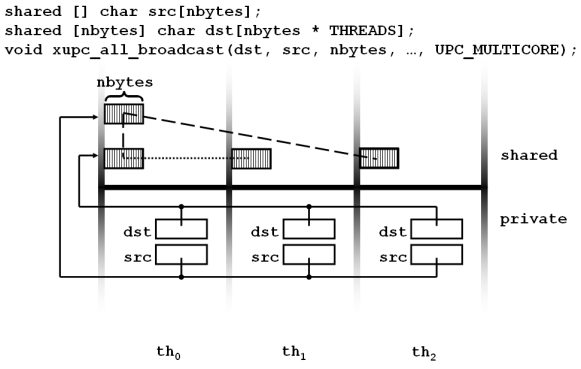

Fig. 3. Ejemplo de `xupc_all_broadcast`

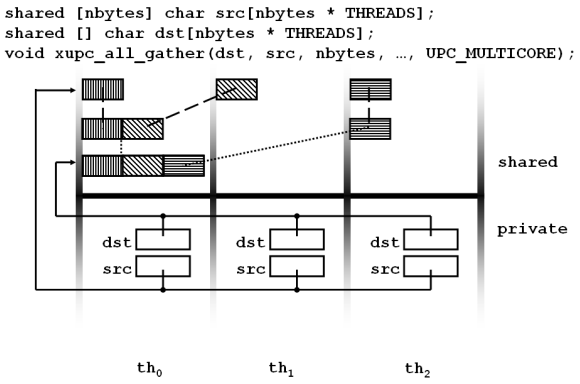

Fig. 4. Ejemplo de `xupc_all_gather`

Nuevas funcionalidades. Se ha implementado la operación de allreduce `xupc_all_reduceT_all` donde `T` representa los 22 tipos de dato definidos definidos en UPC. Estas funciones se han implementado como una llamada a `upc_all_reduceT` seguida de `xupc_all_broadcast`, permiendo la utilización de los tres algoritmos implementados (mediante los flags `UPC_PULL`, `UPC_PUSH` y `UPC_MULTICORE`). BUPC también proporciona esta operación mediante la función `bupc_all_reduceT_all` apoyándose en funcionalidad adicional ofrecida por GASNet.

También se ha desarrollado el conjunto de operaciones `xupc_all_reduceT_mpi`. Estas funciones son semejantes a `MPI_Reduce`, ya que realizan una operación de reducción tal que en lugar de obtener un único valor como `upc_all_reduceT` obtienen un array de tamaño `blk_size` con los resultados de las reducciones en todos los threads. La Figura 5 representa el funcionamiento de esta función.

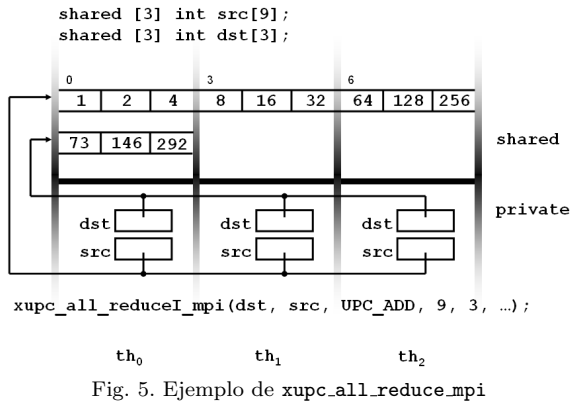

Fig. 5. Ejemplo de `xupc_all_reduce_mpi`

C. Extensión de Colectivas en UPC

La especificación estándar de colectivas en UPC ofrece un conjunto de operaciones con funcionalidad bastante limitada (por ej. regiones de memoria compartida origen y destino disjuntas y operaciones con tamaños de bloque iguales para todos los threads). Asimismo, el número de operaciones definidas es escaso y se carece de algunas de las funcionalidades proporcionadas por MPI, la biblioteca de programación paralela más utilizada en la actualidad.

Asimismo, se han implementado funciones `xupc_all_reduceT_mpi_all` como una llamada a `xupc_all_reduceT_mpi` seguida de `xupc_all_broadcast_in_place` con lo que se admiten los flags `UPC_PULL`, `UPC_PUSH` y `UPC_MULTICORE` para indicar el algoritmo que se desea utilizar para realizar la operación de broadcast *in place*. Estas funciones ofrecen la funcionalidad existente en la biblioteca MPI con `MPI_Allreduce`.

Finalmente, se ha definido una nueva colectiva computacional, `xupc_all_suffix_reduceT`, que

efectúa la operación de sufijo paralelo. En la Figura 6 se representa el funcionamiento de esta función.

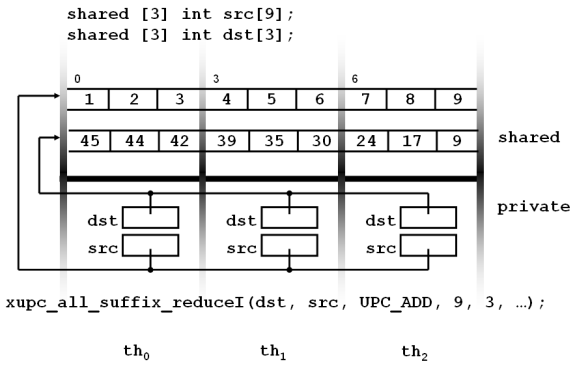

Fig. 6. Ejemplo de `xupc_all_suffix_reduce`

La nueva operación colectiva de movimiento de datos que se ha implementado es `xupc_all_broadcast_in_place`, que realiza una operación de broadcast en un único array en memoria compartida, a semejanza de `MPI_Bcast`. La Figura 7 representa el funcionamiento de esta función.

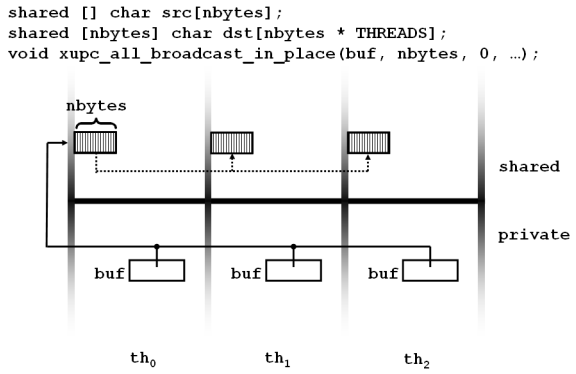

Fig. 7. Ejemplo de `xupc_all_broadcast_in_place`

Colectivas basadas en arrays de valores. Tomando como referencia la propuesta de [12] se ha definido un conjunto de funciones basadas en arrays de valores en la cabecera `xupc_collectiveav.h`. Estas funciones se han implementado como un envoltorio (*wrapper*) sobre las colectivas estándar de UPC, por lo que pueden utilizarse en cualquier implementación que la satisfaga. Las funciones propuestas realizan operaciones semejantes a las definidas en la especificación del lenguaje utilizando arrays privados de valores escalares de tamaño `nelems` en lugar de arrays en el espacio compartido.

La principal motivación para el desarrollo de estas funciones ha sido la mejora de la programabilidad de UPC. Además, trabajar en el espacio de memoria compartido cuando no es necesario acarrea una penalización de rendimiento. No obstante, si se necesita hacer uso de las operaciones colectivas estándar es obligatorio utilizar vectores que se encuentren almacenados en el espacio de memoria compartido.

Como se puede apreciar en la Tabla I, la utilización de las funciones que componen esta extensión liberan

Código UPC estándar	Código `xupc_collectiveav.h`
declarar array privado	declarar array privado
uso del array	uso del array
reserva de espacio shared	
copiar datos a espacio shared	
upc_all_...	xupc_allav_...
copiar resultado array privado	
liberar espacio shared	
uso de array privado (resultado)	uso de array privado (resultado)

TABLA I

`XUPC_COLLECTIVEAV.H`: COMPARATIVA DE USO

significativamente al programador de la escritura de código, disminuyendo con ello el esfuerzo necesario y las posibilidades de cometer errores.

Vector Variant Collectives. En [13] también se han definido operaciones colectivas que permiten que cada bloque de datos involucrado en la operación tenga un tamaño diferente, así como la utilización de bloques no contiguos. Siguiendo esta propuesta se han implementado funciones de broadcast, gather y allgather.

La flexibilidad a la hora de indicar dónde se copian los datos en cada thread se logra mediante la utilización de un array de desplazamientos `ddisp`. Así, el programador indicará en `ddisp[i]` la posición dónde quiere que comiencen a escribirse los datos correspondientes al thread i. La Figura 8 representa el funcionamiento de la función `xupc_all_broadcast_v`.

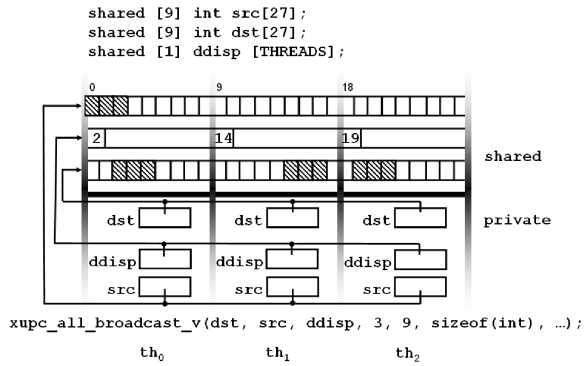

Fig. 8. Ejemplo de `xupc_all_broadcast_v`

La función `xupc_all_gather_v` copia un bloque de `nelems[i]` elementos en memoria compartida apuntado por `src`, con desplazamiento `sdisp[i]`, a una zona de memoria compartida que tiene afinidad con un único thread, apuntada por `dst`, más un desplazamiento `ddisp[i]`. La Figura 9 representa el funcionamiento de esta función. Con esta misma idea, también se ha implementado la función `xupc_all_gather_all_v`, que permite que todos los threads dispongan del resultado de la operación `xupc_all_gather_v` en una zona de memoria con afinidad a cada uno.

D. Impacto en la Programabilidad

UPC permite mejorar la programabilidad en aplicaciones paralelas [14] ya que se logran implementaciones claras, concisas y menos complejas que las desarrolladas con otros paradigmas alternativos. A

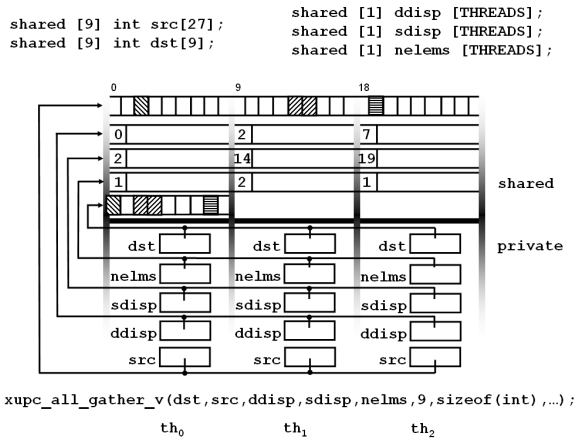

Fig. 9. Ejemplo de `xupc_all_gather_v`

esta mejora de la programabilidad contribuyen especialmente las operaciones colectivas, ya que reducen la cantidad de código necesario a la vez que proporcionan una mayor sencillez conceptual. Además, la reutilización de código disminuye la probabilidad de cometer errores a la hora de codificar. La biblioteca de operaciones colectivas propuesta incrementa la productividad de los códigos UPC al aumentar su eficiencia, a la vez que amplía las funcionalidades disponibles, aumentando la programabilidad en UPC.

IV. Evaluación del Rendimiento

Se ha evaluado experimentalmente el rendimiento de la biblioteca desarrollada en el supercomputador Finis Terrae [15] del Centro de Supercomputación de Galicia (CESGA), compuesto por 142 nodos, cada uno con 8 procesadores Intel IA64 Itanium2 Montvale Dual Core a 1,6 GHz (16 cores por nodo) y 128 GB de memoria. Estos nodos se encuentran interconectados mediante Infiniband 4x DDR a 20 Gbps. El compilador de UPC es Berkeley 2.6.0, utilizando además el Intel C Compiler 10.

En la Figura 10 se muestra el ancho de banda para distintas implementaciones de `upc_all_gather`, la de BUPC y los algoritmos de la biblioteca propuesta (XUPC) (`UPC_PULL`, `UPC_PUSH` y `UPC_MULTICORE`). Se ha evaluado su rendimiento sobre InfiniBand (conducto `ibv` de BUPC) y memoria compartida (conducto `smp` de BUPC). En ambos casos se utiliza la biblioteca `pthreads` para ejecutar `pth` threads de UPC en cada uno de los `N` nodos que se están utilizando.

Así, con `pth`=16 y N=1, 16 threads en un único nodo con el conducto `smp`, los algoritmos implementados en XUPC ofrecen un mejor rendimiento que la versión proporcionada por defecto (BUPC). De hecho, `UPC_PUSH` maximiza el rendimiento debido al inherente mayor paralelismo en la situación en que todos los threads envían sus datos al thread destino. En cambio, con 4 nodos y 16 threads por nodo (en total 64 threads), vemos como la mejor opción es la proporcionada por el algoritmo Multi-core Aware.

La Figura 11 muestra el ancho de banda de distintas versiones de `upc_all_gather_all`. En memoria compartida (`smp`) y con tamaños de transferencia pequeños `UPC_PULL` es la mejor implementación. No obstante, a medida que aumenta el tamaño de los datos a transferir la versión `UPC_PUSH` llega a obtener los mayores rendimientos. En la ejecución en 4 nodos, el algoritmo Multi-core Aware y la implementación BUPC obtienen los mejores resultados.

En la Figura 12 se muestra el ancho de banda de distintas versiones de `xupc_all_broadcast_in_place`. En la ejecución en un único nodo, `UPC_PULL` es el algoritmo más eficiente, mientras que para la ejecución en distintos nodos la opción más eficiente es `UPC_MULTICORE`.

Por norma general, se aprecia que la biblioteca desarrollada (XUPC) se comporta mejor que la biblioteca de colectivas de BUPC. Como era de esperar, la utilización del algoritmo Multi-core Aware penaliza el rendimiento en memoria compartida, mientras que en un entorno de clúster multi-core incrementa significativamente el rendimiento de los algoritmos pull y push. Otro punto a destacar es la conveniencia de utilizar la versión pull (push en el caso de gather) cuando se esté utilizando un único nodo debido a su inherente paralelismo. Esto nos lleva a considerar que en el caso de que fuese posible utilizar barreras de sincronización a nivel de nodo podría ser adecuado utilizar un algoritmo pull para la operativa intranodo en el algoritmo Multi-core Aware.

V. Conclusiones

Este artículo presenta una biblioteca de primitivas colectivas para UPC que proporciona diversos algoritmos que permiten explotar la localidad de los datos en UPC, un aspecto de crucial importancia en la ejecución de lenguajes PGAS sobre arquitecturas clúster multi-core. Además, la biblioteca desarrollada extiende el estándar de operaciones colectivas de UPC proporcionando nuevas operaciones colectivas que solventan sus principales inconvenientes. Así, ha sido posible mejorar la programabilidad y eficiencia de las operaciones colectivas en UPC, incrementando de este modo su productividad, como ha sido evaluada experimentalmente en un clúster multi-core.

Agradecimientos

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

Referencias

[1] UPC Consortium, "UPC Language Specifications v1.2," Tech. Rep., Lawrence Berkeley National Lab, May 2005.

[2] E. Wiebel, D. Greenberg, and S. R. Seidel, "UPC Collective Operations Specifications v1.0," `http://www.gwu.edu/~upc/docs/UPC_Coll_Spec_V1.0.pdf`.

[3] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn, "Collective Communication: Theory, Practice and Experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[4] R. Nishtala, G. Almasi, and C. Cascaval, "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, 2008, pp. 99–110.

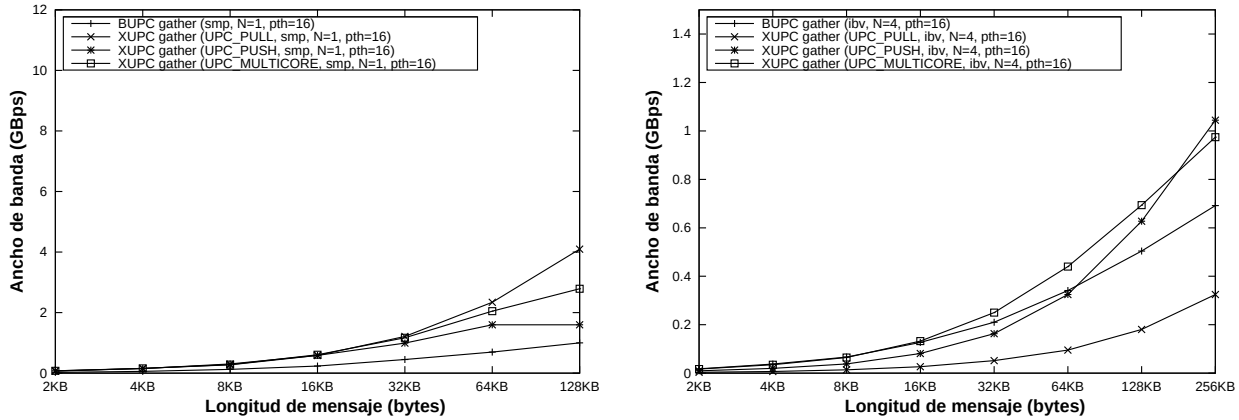

Fig. 10. Ancho de banda de `xupc_all_gather` con conductos `smp` e `ibv`

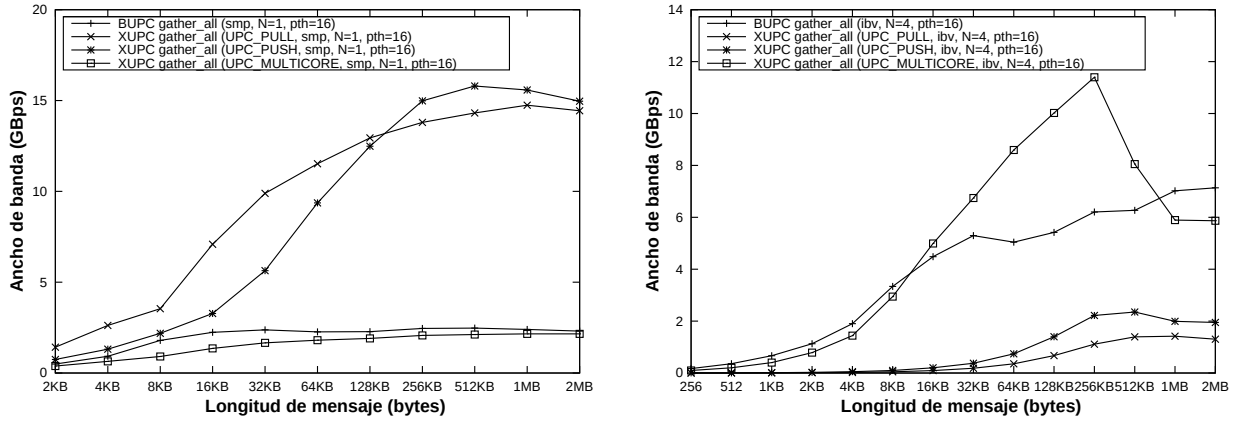

Fig. 11. Ancho de banda de `xupc_all_gather_all` con conductos `smp` e `ibv`

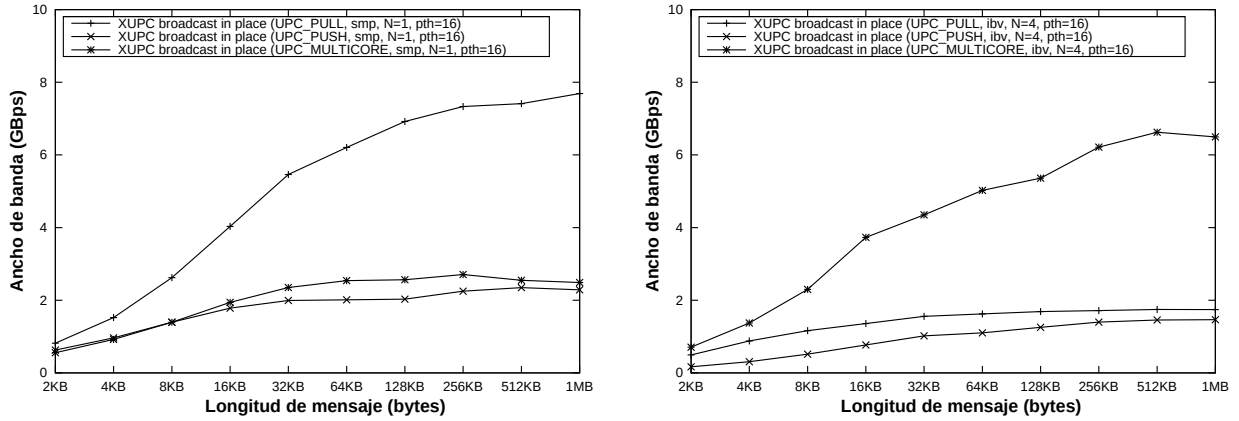

Fig. 12. Ancho de banda de `xupc_all_broadcast_in_place` con conductos `smp` e `ibv`

[5] D. Bonachea and R. Nishtala, "UPC Teams," `https://upc-wiki.lbl.gov/UPC/index.php/Private:UPC_Teams`.

[6] Z. Ryne and S. R. Seidel, "Ideas and Specifications for the new One-sided Collective Operations in UPC," `http://www.upc.mtu.edu/papers/OnesidedColl.pdf`.

[7] Z. Ryne and S. R. Seidel, "A Specification of The Extensions to The Collective Operations of Unified Parallel C," Tech. Rep., Michigan Technological University, 2005.

[8] D. Bonachea, "Proposal for Extending the UPC Memory Copy Library Functions, v2.0," `http://upc.lbl.gov/publications/upc_memcpy.pdf`.

[9] UC Berkeley / LBNL, "GASNet (Global-Address Space Networking)," `http://gasnet.cs.berkeley.edu/`.

[10] UC Berkeley / LBNL, "Berkeley Unified Parallel C (UPC) Project," `http://upc.lbl.gov/`.

[11] Michigan Technological University, "UPC Collectives Reference Implementation," `http://www.upc.mtu.edu/collectives/col1.html`.

[12] D. Bonachea, "UPC Collectives Value Interface, v1.2," `http://upc.lbl.gov/docs/user/README-collectivev.txt`.

[13] Z. Ryne and S. R. Seidel, "UPC Extended Collective Operations Specification," `http://www.upc.mtu.edu/papers/UPC_CollExt.pdf`.

[14] F. Cantonnet, Y. Yao, M. M. Zahran, and T. A. El-Ghazawi, "Productivity Analysis of the UPC Language," in *Proc. 18th Intl. Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe (NM), 2004.

[15] TOP500, "Supercomputador Finis Terrae," `http://www.top500.org/system/9156/`.