UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations

Jorge González Domínguez

Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems

University of A Coruña, Spain

PhD Thesis

UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations

Jorge González Domínguez

November 2012

PhD Advisors:
María José Martín Santamaría
Juan Touriño Domínguez

Dra. María José Martín Santamaría	Dr. Juan Touriño Domínguez
Profesora Titular de Universidad	Catedrático de Universidad
Dpto. de Electrónica y Sistemas	Dpto. de Electrónica y Sistemas
Universidad de A Coruña	Universidad de A Coruña

CERTIFICAN

Que la memoria titulada "*UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations*" ha sido realizada por D. Jorge González Domínguez bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidad de A Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 19 de Noviembre de 2012

Fdo.: María José Martín Santamaría	Fdo.: Juan Touriño Domínguez
Directora de la Tesis Doctoral	Director de la Tesis Doctoral

Fdo.: Jorge González Domínguez
Doctorando

Resumen de la Tesis

Introducción

La necesidad de una potencia de cómputo creciente por parte de aplicaciones de diversos campos de la ciencia y la ingeniería, o incluso del ocio, ha sido la impulsora de la evolución del computador. La aproximación tradicional para esta evolución ha sido la mejora de tecnologías ya conocidas. Sin embargo, esta aproximación está llegando a su límite físico. Una nueva era ha empezado, caracterizada por los procesadores multinúcleo que habilitan la ejecución de aplicaciones multihilo. La popularidad de los sistemas multinúcleo ha hecho que la programación paralela llegue a ser una tarea de vital importancia para explotar de forma efectiva el hardware y aumentar el rendimiento de las aplicaciones.

La programación paralela se realiza tradicionalmente mediante la utilización de bibliotecas de paso de mensajes en memoria distribuida o de construcciones de programación concurrente (barreras, semáforos ...) en memoria compartida. Hoy en día Message Passing Interface (MPI) y Open MultiProcessing (OpenMP) constituyen los estándares de facto para la programación en arquitecturas de memoria distribuida y compartida, respectivamente. Recientemente ha surgido un nuevo modelo de programación, el modelo Partitioned Global Address Space (PGAS), que proporciona importantes ventajas sobre los modelos tradicionales. En el modelo PGAS todos los hilos comparten un espacio de memoria como en el modelo de memoria compartida. Sin embargo, este espacio de memoria está lógicamente particionado entre los diferentes hilos como en el modelo de memoria distribuida. Los lenguajes PGAS consiguen el equilibrio perfecto entre facilidad de uso y explotación de la localidad de los datos. En los últimos años el modelo PGAS ha estado ganando popularidad, siendo ejemplos representativos de este modelo los lenguajes Unified Parallel C (UPC), Titanium y Co-Array Fortran (CAF).

UPC es un lenguaje de programación paralela que surge como una extensión a la especificación estándar del lenguaje C y que está ganando interés y popularidad dentro de la comunidad de High Performance Computing (HPC). UPC fue originalmente diseñado para computadores paralelos de gran escala y entornos clúster. Sin embargo, su modelo de programación PGAS lo convierte en una excelente opción para sistemas multinúcleo, en los cuales la memoria principal es físicamente compartida. Hoy en día existen compiladores de UPC disponibles para la mayoría de los sistemas paralelos de los principales fabricantes, así como investigación activa en muchas universidades y centros de investigación en supercomputación. A través de medidas experimentales exhaustivas desarrolladas en trabajos previos, se ha demostrado la viabilidad de UPC frente a otros paradigmas de programación paralela habituales. Además, se ha demostrado también que los códigos UPC pueden conseguir buena escalabilidad en hasta cientos de miles de procesadores siempre que cuenten con el adecuado soporte por parte del compilador y del sistema runtime. Sin embargo, una barrera para una mayor aceptación de UPC por parte de los usuarios es la carencia de bibliotecas de apoyo en este lenguaje para los desarrolladores de algoritmos y aplicaciones.

Las rutinas del conjunto Basic Linear Algebra Subprograms (BLAS) proporcionan bloques estándar para realizar operaciones básicas con vectores y matrices. Son utilizadas por científicos e ingenieros para conseguir buenos rendimientos en arquitecturas monoprocesador. Las rutinas SparseBLAS son una versión reducida de las BLAS para matrices y vectores dispersos, los cuales introducen patrones de acceso irregulares. Las Parallel Basic Linear Algebra Subprograms (PBLAS) y las Parallel Sparse Basic Linear Algebra Subprograms (PSBLAS) son una implementación de las BLAS y las SparseBLAS, respectivamente, para computación paralela, las cuales han sido desarrolladas para facilitar a los usuarios la programación en arquitecturas de memoria distribuida. Sin embargo, no existen versiones paralelas en UPC.

En la actualidad existe una creciente tendencia a desarrollar códigos que tengan en cuenta el sistema en el que están siendo ejecutados para realizar optimizaciones de forma automática. Entre las diferentes arquitecturas paralelas, la optimización de aplicaciones en los clusters de sistemas multinúcleo presenta un importante desafío ya que poseen una arquitectura de memoria híbrida (distribuida/compartida) con latencias de comunicaciones no uniformes. Además, accesos concurrentes a la misma caché o al mismo módulo de memoria desde distintos núcleos pueden conducir a una pérdida de ancho de banda a memoria.

Esta Tesis Doctoral, *“UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations”*, nace del intento de impulsar la utilización y expansión de los lenguajes PGAS (especialmente de UPC) aportando a los actuales y futuros programadores una biblioteca de computación numérica paralela, UPCBLAS. La biblioteca contiene un subconjunto de rutinas BLAS (las más utilizadas por sus usuarios) con una interfaz basada en las características de UPC y centrada en aumentar la programabilidad de este lenguaje, sin comprometer excesivamente por ello el rendimiento. Precisamente con el fin de mejorar el rendimiento de las rutinas numéricas, esta Tesis Doctoral también ha llevado al desarrollo de Servet, una herramienta que detecta mediante benchmarks de forma automática un conjunto de parámetros hardware muy útiles para optimizar el rendimiento de los códigos paralelos. De hecho, las rutinas de UPCBLAS usarán automáticamente la interfaz proporcionada por Servet para reducir sus tiempos de ejecución. Por último, esta Tesis proporciona una evaluación experimental detallada de la biblioteca, así como ejemplos prácticos de su uso.

Metodología de Trabajo

La metodología de trabajo seguida en el desarrollo de la presente Tesis Doctoral ha consistido en:

- Definir la lista de tareas a realizar durante la Tesis, teniendo en cuenta los trabajos previos y los recursos disponibles.
- Determinar su secuencia u orden de ejecución, ateniéndose a las restricciones que pudiesen existir y al orden más favorable.
- Establecer su duración y la oportunidad de su desarrollo en un momento determinado.

- Organizar las acciones o tareas por bloques de cierta entidad que definan etapas.
- Definir, para cada etapa, las metas a alcanzar (u objetivos concretos a lograr en tiempo definido), sabiendo que en cada etapa puede haber una o varias metas.

Tareas

De este modo, la lista de tareas (*Tn.m*), agrupadas en bloques (**Bn**), desarrolladas en la presente Tesis han sido:

B1 Estudio del estado del arte relacionado con bibliotecas numéricas en UPC.

T1.1 Búsqueda de bibliotecas numéricas desarrolladas con anterioridad para UPC. Se determina que no existen trabajos previos basados en el desarrollo de bibliotecas numéricas en UPC o cualquier otro lenguaje PGAS.

T1.2 Estudio de otros trabajos previos relacionados con computación numérica paralela en UPC. Estos trabajos abarcan principalmente productos de matrices y factorizaciones (LU, Cholesky...). Se recopilan una serie de técnicas de programación que mejoran el rendimiento de los códigos escritos en este lenguaje para su posterior uso en la implementación de la biblioteca.

T1.3 Análisis de bibliotecas numéricas paralelas que siguen otro paradigma de programación y estudio de su posible adaptación a los lenguajes PGAS.

B2 Diseño e implementación de las rutinas densas.

T2.1 Selección de las funciones a incluir en la parte de computación densa de la biblioteca.

T2.2 Diseño de la interfaz de las rutinas y especificación de un método lógico de nomenclatura de las funciones para facilitar su uso a los programadores.

T2.3 Desarrollo de los tipos de datos necesarios en las rutinas, tanto los enumerados para reflejar distintas opciones del problema (distribución por filas/columnas, matrices triangulares inferiores/superiores, etc.) como las estructuras para tratar con tipos de datos complejos.

T2.4 Implementación de una primera versión de las rutinas numéricas seleccionadas. Durante este proceso se hace especial hincapié en incluir técnicas de optimización cuya eficiencia en códigos UPC está demostrada en el estado del arte.

T2.5 Confección de un mecanismo de configuración y compilación que sea sencillo de utilizar por los usuarios.

T2.6 Testeo de que las rutinas numéricas incluidas en la biblioteca proporcionan los resultados correctos.

B3 Evaluación del rendimiento de la biblioteca y optimización.

T3.1 Implementación de un conjunto de benchmarks que permitan conocer los speedups y eficiencias de las diferentes rutinas, tanto para escalado fuerte como débil.

T3.2 Ejecución de dichos benchmarks en el supercomputador Finis Terrae, del Centro de Supercomputación de Galicia (CESGA).

T3.3 Estudio de los resultados obtenidos, prestando especial atención a la detección de aquellas partes del código que puedan ser optimizables mediante un mejor conocimiento de la arquitectura.

T3.4 Desarrollo de Servet, un conjunto de microbenchmarks que obtienen una serie de parámetros hardware interesantes para la optimización de programas paralelos. Esta herramienta debe proporcionar al menos las características hardware que se determinaron en la tarea anterior como necesarias para optimizar las rutinas.

T3.5 Inclusión de técnicas de optimización del rendimiento basadas en los parámetros hardware proporcionados por Servet.

T3.6 Evaluación de nuevo del rendimiento usando los mismos benchmarks en máquinas distintas, para comprobar que las optimizaciones implementadas en la tarea anterior son válidas para diferentes arquitecturas y compiladores.

B4 Estudio de la utilidad de la biblioteca en un entorno real.

- *T4.1* Determinación de un conjunto de códigos numéricos que pueden implementarse usando UPCBLAS.
- *T4.2* Implementación de dichos códigos paralelos usando las rutinas de la biblioteca.
- *T4.3* Análisis de la mejora de programabilidad y productividad obtenida gracias a UPCBLAS.
- *T4.4* Detección y posterior inclusión de posibles mejoras en el diseño de la biblioteca de cara a aumentar la programabilidad.
- *T4.5* Desarrollo y ejecución de benchmarks que muestren el rendimiento de los códigos paralelos implementados con el apoyo de la biblioteca.

B5 Desarrollo de las rutinas dispersas.

- *T5.1* Diseño de la interfaz de las rutinas dispersas, intentando reutilizar, cuando sea posible, elementos que han sido probados útiles para computación densa.
- *T5.2* Selección de las rutinas a implementar y los formatos de almacenamiento para matrices dispersas a utilizar.
- *T5.3* Implementación de dichas rutinas con dichos formatos. De nuevo, se intentan incluir las técnicas de optimización de la parte densa (tarea *T2.4*)
- *T5.4* Comprobación de que el resultado de las rutinas dispersas es el correcto mediante la ejecución de tests.
- *T5.5* Uso de benchmarks para evaluar el rendimiento de las rutinas dispersas.

B6 Determinación de las principales conclusiones y líneas de trabajo futuras.

- *T6.1* Determinación de las principales conclusiones.
- *T6.2* Evaluación de las principales líneas de investigación abiertas a raíz del trabajo desarrollado.
- *T6.3* Redacción de la memoria final de la Tesis Doctoral.

El trabajo llevado a cabo en estas tareas ha sido recogido en la presente memoria. Así, las tareas del primer bloque han sido recogidas en el Capítulo 1. El segundo bloque constituye el Capítulo 2. Las tareas del tercer y cuarto bloque están contenidas en los Capítulos 3 y 4. El quinto bloque constituye el Capítulo 5. Finalmente, el sexto bloque está incluido dentro del último capítulo de conclusiones y trabajo futuro.

Metas

Asimismo, la lista de metas (*Mn.m*) asociadas con cada bloque (**Bn**) de la Tesis Doctoral han sido:

B1 Estudio del estado del arte relacionado con bibliotecas numéricas en UPC.

- *M1.1* Evaluación del estado actual de los trabajos relacionados con computación numérica paralela para lenguajes PGAS.
- *M1.2* Obtención de aquellas ideas de diseño e implementación que puedan ser útiles para el desarrollo de la biblioteca numérica en UPC.

B2 Diseño e implementación de las rutinas densas.

- *M2.1* Conjunto de rutinas para computación densa que exploten las ventajas del paradigma de programación PGAS en general y de UPC en particular. Esta primera implementación debe estar focalizada en la mejora de la programabilidad y la productividad de los potenciales usuarios.

B3 Evaluación del rendimiento de la biblioteca y optimización.

- *M3.1* Herramienta que permita obtener de forma automática ciertos parámetros hardware válidos para optimizar códigos paralelos. Esta herramienta debe proporcionar al menos aquellos parámetros necesarios para optimizar el rendimiento de las rutinas implementadas para *M2.1*.
- *M3.2* Mismo conjunto de rutinas que en *M2.1* optimizado haciendo uso del conocimiento de las características hardware proporcionadas por la herramienta de *M3.1*.
- *M3.3* Conjunto de resultados experimentales para la evaluación del redimiento de las rutinas de *M3.2*.

B4 Estudio de la utilidad de la biblioteca en un entorno real.

M4.1 Códigos numéricos de más alto nivel que hagan uso por debajo de la biblioteca de *M3.2*.

M4.2 Evaluación en términos de aumento de rendimiento, programabilidad y productividad de la biblioteca en dichos códigos.

B5 Desarrollo de las rutinas dispersas.

M5.1 Conjunto de rutinas para computación dispersa.

M5.2 Estudio de la adecuación de distintos formatos de almacenamiento para matrices dispersas a la computación numérica en UPC.

B6 Determinación de las principales conclusiones y líneas de trabajo futuras.

M6.1 Memoria final de la Tesis Doctoral que recoge las principales conclusiones y líneas futuras de investigación.

Medios

Los medios necesarios para realizar esta Tesis Doctoral, siguiendo la metodología de trabajo anteriormente descrita, han sido los siguientes:

- Material de trabajo y financiación económica proporcionados fundamentalmente por el Grupo de Arquitectura de Computadores de la Universidad de A Coruña y el Ministerio de Educación y Ciencia (beca FPU AP2008-01578).
- Además, esta Tesis se ha financiado a través de los siguientes proyectos de investigación:
  - De financiación internacional a través del proyecto *"Improving UPC Usability and Performance in Constellation Systems: Implementation/Extension of UPC Libraries"*, suscrito con Hewlett-Packard S.L. (HP).
  - De financiación estatal (Ministerio de Educación y Ciencia y Ministerio de Ciencia e Innovación) a través de los proyectos TIN2007-67537-C03-02, TIN2010-16735 y TIN2010-12011-E.

- Acceso a material bibliográfico, a través de la biblioteca de la Universidad de A Coruña.

- Acceso a clusters con múltiples procesadores/núcleos por nodo:

  - Supercomputador *Finis Terrae* (CESGA, 2008-actualidad). 144 nodos con 16 núcleos de procesador Intel Itanium2 Montvale a 1.6 GHz interconectados mediante InfiniBand, además de contar con un sistema Superdome de memoria compartida con 128 núcleos Itanium2 Montvale a 1.6 GHz y 1 TB de RAM.

  - Clúster *Plutón* (Universidad de A Coruña, 2009-actualidad). 16 nodos con 16 núcleos de procesador Intel Xeon Nehalem a 2.26 GHz interconectados mediante InfiniBand.

  - Supercomputador *Carver* (NERSC, 2010-actualidad). 1120 nodos con 8 núcleos de procesador Intel Xeon Nehalem a 2.67 GHz interconectados mediante InfiniBand.

  - Supercomputador *Hopper* (NERSC, 2011-actualidad). 6384 nodos AMD Magny-Cours con 24 núcleos a 2.1 GHz interconectados mediante Gemini, una red propietaria de Cray con una estructura de Toro 3D.

  - Supercomputador *HECToR* (EPCC, 2011-actualidad). 2816 nodos con dos procesadores AMD Interlagos de 16 núcleos cada uno (32 núcleos por nodo) a 2.3 GHz interconectados también a través de una red Gemini.

- Una ayuda para una estancia de investigación de 13 semanas en 2011 en el Lawrence Berkeley National Laboratory (LBNL) en Berkeley, Estados Unidos, obtenida en concurrencia competitiva en una convocatoria del Ministerio de Educación y Ciencia. Esta estancia permitió desarrollar el bloque de tareas *B4* así como establecer una relación de colaboración con el grupo de trabajo de Berkeley UPC de la Profesora Katherine Yelick, el más importante del mundo dedicado a la investigación en el ámbito del lenguaje UPC. Esta relación permitirá, por un lado, la distribución de la biblioteca objeto de esta Tesis junto con el compilador Berkeley UPC y, por otro lado, el desarrollo conjunto de trabajos de investigación sobre computación numérica de altas prestaciones usando UPC.

- Una ayuda para una estancia de investigación de 9 semanas en 2012 en el Edinburgh Parallel Computing Center (EPCC) en Edimburgo, Reino Unido, obtenida mediante concurrencia competitiva en el programa HPC-Europa2. La realización de esta estancia permitió acceder al supercomputador *HECToR* y trabajar con los investigadores de dicho centro para obtener parte de los resultados experimentales de la tarea *T.3.6*.

Conclusiones

Esta Tesis Doctoral, *"UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations"*, ha permitido el desarrollo de UPCBLAS, una biblioteca numérica paralela para Unified Parallel C. UPC sigue el paradigma de programación PGAS, el cual, comparado con el paradigma de paso de mensajes, permite aumentar la programabilidad sin hipotecar el rendimiento gracias a la explotación de la localidad de datos. Los lenguajes PGAS son una alternativa muy interesante para la programación de sistemas paralelos, especialmente para aquellos con una arquitectura de memoria híbrida como los clusters de sistemas multinúcleo.

Sin embargo, la falta de bibliotecas limita la productividad de los programadores de lenguajes PGAS. Por ejemplo, el análisis del estado del arte reveló que no existía absolutamente ninguna biblioteca numérica paralela para UPC. Por tanto, los programadores que necesitaban, por ejemplo, rutinas de las interfaces BLAS y SparseBLAS, debían utilizar bibliotecas basadas en el paradigma de paso de mensajes. Se identificó que el mayor problema de estas bibliotecas era su complejidad de uso, algo que puede solventarse explotando directamente las cualidades de los lenguajes PGAS en general y de UPC en particular. La biblioteca desarrollada en esta Tesis proporciona funciones para computación numérica densa y dispersa que, siguiendo el modelo de programación PGAS, facilita considerablemente su uso con respecto a otras bibliotecas diseñadas para otros paradigmas de programación.

Pero el objetivo de la biblioteca no consiste simplemente en facilitar la programación a sus usuarios sino también en que obtengan un buen rendimiento. Para ello las rutinas incluyen ciertas técnicas de optimización. Unas fueron identificadas mediante el estudio del estado del arte y otras fueron desarrolladas ad hoc basándose en los parámetros hardware de la máquina donde se ejecutan las rutinas. Para poder implementar estas últimas técnicas se ha creado Servet, una herramienta que, por medio de benchmarks, permite obtener de forma automática los parámetros hardware necesarios.

El rendimiento de la biblioteca se ha evaluado exhaustivamente y se ha comparado con otras bibliotecas numéricas en dos supercomputadores completamente diferentes, no solo por su tamaño y arquitectura, sino también por el compilador de UPC empleado. Los resultados experimentales demuestran que la facilidad de uso no implica necesariamente un rendimiento muy inferior al de las bibliotecas basadas en el paradigma de paso de mensajes.

Para concluir, conviene destacar que el uso de los mecanismos del lenguaje UPC para distribuir matrices conlleva un aumento de la programabilidad pero también limita los tipos de distribuciones que se pueden utilizar a una sola dimensión (por filas o por columnas) y con un tamaño de distribución fijo. Aunque esta biblioteca proporciona un buen rendimiento, probablemente sería posible mejorarlo mediante el uso de otro tipo de distribuciones con un tamaño de bloque variable o distribuciones de datos 2D, 3D o 2.5D. Sin embargo, siempre habrá que tener en cuenta que si nuevas extensiones del lenguaje UPC no incluyen estructuras que soporten este tipo de distribuciones, serán los propios desarrolladores de bibliotecas quienes deban implementar dichas estructuras con una sintaxis tal que no se aumente la complejidad de uso de dichas bibliotecas.

Principales Contribuciones

Las principales aportaciones de esta Tesis son:

- La biblioteca numérica paralela UPCBLAS [48, 49, 50, 51], que proporciona un subconjunto de rutinas BLAS con una interfaz que explota las características del paradigma PGAS para proporcionar un buen rendimiento mejorando ostensiblemente la programabilidad con respecto a otras bibliotecas numéricas.
- Servet, una herramienta que permite obtener de forma automática mediante benchmarks una serie de parámetros hardware muy útiles para optimizar pro-

gramas paralelos [52, 53], incluyendo una evaluación de su impacto positivo a la hora de optimizar las rutinas numéricas de UPCBLAS [51].

- Un estudio del uso de UPCBLAS para facilitar el desarrollo de algoritmos numéricos paralelos más complejos, haciendo énfasis en el rendimiento y la programabilidad [47].
- La implementación de un conjunto de rutinas SparseBLAS en UPC, estudiando su comportamiento de acuerdo al formato de almacenamiento de matrices dispersas utilizado [45, 46].
- La evaluación en UPC de nuevos y complejos algoritmos que combinan el solapamiento de comunicación y computación junto con la minimización de comunicaciones para el ámbito de la computación numérica de forma que, en el futuro, puedan incorporarse en las rutinas de UPCBLAS [44].

Publications from the Thesis

Journal Papers (3)

- J. González-Domínguez, O. García-López, G. L. Taboada, M. J. Martín, and J. Touriño. Performance Evaluation of Sparse Matrix Products in UPC. *The Journal of Supercomputing,* 2012 (In press).
- J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan. UPCBLAS: A Library for Parallel Matrix Computations in Unified Parallel C. *Concurrency and Computation: Practice and Experience*, 24(14):1645-1667, 2012
- J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Automatic Mapping of Parallel Applications on Multicore Architectures using the Servet Benchmark Suite. *Computers and Electrical Engineering*, 32(2):258-269, 2012.

International Conferences (7)

- E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *Proc. 24th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, 2012.
- J. González-Domínguez, O. A. Marques, M. J. Martín, G. L. Taboada, and J.

Touriño. Design and Performance Issues of Cholesky and LU Solvers using UPCBLAS. In *Proc. 10th IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA'12)*, Leganés, Spain, 2012.

- J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan. A Library for Parallel Numerical Computation in UPC. Poster in *Proc. 23rd ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'11)*, Seattle, WA, USA, 2011.

- J. González-Domínguez, O. García-López, G. L. Taboada, M. J. Martín, and J. Touriño. SparseBLAS Products in UPC: An Evaluation of Storage Formats. In *Proc. 11th Intl. Conf. on Computational and Mathematical Methods in Science and Engineering (CMMSE'11)*, Benidorm, Spain, 2011.

- J. González-Domínguez, M. J. Martín, G. L. Taboada, and J. Touriño. Dense Triangular Solvers on Multicore Clusters using UPC. In *Proc. 11th Intl. Conf. on Computational Science (ICCS'11)*, Singapore, 2011.

- J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In *Proc. 24th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'10)*, Atlanta, GA, USA, 2010.

- J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, and A. Gómez. A Parallel Numerical Library for UPC. In *Proc. 15th Intl. European Conf. on Parallel and Distributed Computing (Euro-Par'09)*, Delft, The Netherlands, 2009.

National Conferences (2)

- J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Servet: Una Suite de Benchmarks para el Soporte del Autotuning en Clusters de Sistemas Multinúcleo. In *Actas de las XXI Jornadas de Paralelismo*, Valencia, Spain, 2010.

- J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, and A. Gómez. Una Biblioteca Numérica Paralela para UPC. In *Actas de las XX Jornadas de Paralelismo*, A Coruña, Spain, 2009.

Abstract

The popularity of Partitioned Global Address Space (PGAS) languages has increased during the last years thanks to their high programmability and performance through an efficient exploitation of data locality, especially on hierarchical architectures like multicore clusters. This PhD Thesis describes UPCBLAS, a parallel library for numerical computation using the PGAS Unified Parallel C (UPC) language. The routines are built on top of sequential BLAS and SparseBLAS functions and exploit the particularities of the PGAS paradigm, taking into account data locality in order to achieve a good performance. However, the growing complexity in computer system hierarchies due to the increase in the number of cores per processor, levels of cache (some of them shared) and the number of processors per node, as well as the high-speed interconnects, demands the use of new optimization techniques and libraries that take advantage of their features. For this reason, this Thesis also presents Servet, a suite of benchmarks focused on detecting a set of parameters with high influence on the overall performance of multicore systems. UPCBLAS routines use the hardware parameters provided by Servet to implement optimization techniques that improve their performance. The performance of the library has been experimentally evaluated on several multicore supercomputers and compared to message-passing-based parallel numerical libraries, demonstrating good scalability and efficiency. UPCBLAS has also been used to develop more complex numerical codes in order to demonstrate that it is a good alternative to MPI-based libraries for increasing the productivity of numerical application developers.

A mis padres

A mi hermanita

A María, mi niña

Acknowledgments

This Thesis is not the result only of my own effort; there are many people involved in this work whose support and dedication I want to acknowledge. First and foremost, I want to acknowledge my PhD advisors María and Juan for the confidence they placed in me, and the support they gave me during all these years. I also want to thank Guillermo because he has been a support and reference for me since the beginning of this PhD Thesis. I cannot forget my other colleagues in the Computer Architecture Group because they made my experience in the lab easier during all these years, especially Sabela, Jose and Raquel.

I gratefully thank my parents, who have always been there for me. I also thank my sister Loreto, my girlfriend María and my friends Jorge, Jaime, Lucía, Carlos, Vero, Bea, Kike, Pablo, Paula, Raúl and Dani, who have always believed even more than me in what I was doing, and whose support helped me to overcome the doubts that all PhD students have about if their effort is worthy. Besides, I thank my hockey coaches, teammates and pupils for helping me to entertain my mind out of the development of this Thesis and for understanding my mood swings.

I want to acknowledge Katherine Yelick and Michele Weiland for hosting me during my visits to the LBNL (Lawrence Berkeley National Laboratory) and EPCC (Edinburgh Parallel Computing Center), respectively. I thank Yili Zheng, Osni A. Marques, Evangelos Georganas and Catherine Inglis for their valuable help during those visits. I also thank Enrique Quintana-Ortí for all his comments to improve this Thesis and Tomás Fernández Pena and Damián Álvarez Mallón for their help in the development of UPCBLAS.

I gratefully thank CESGA (Galicia Supercomputing Center), NERSC (National Energy Research Scientific Computing Center) and EPCC for providing access to the Finis Terrae, Carver, Hopper and HECToR supercomputers.

Last, but not least, I am thankful to the following institutions for funding this work: the Computer Architecture Group and the Department of Electronics and

Systems for the human and material support, the University of A Coruña for giving me the opportunity of starting a career in academia and for funding my attendance at some conferences, Hewlett Packard S.L. for the project *"Improving UPC Usability and Performance in Constellation Systems: Implementation/Extension of UPC Libraries"*, the Pan-European Research Infrastructure for High Performance Computing for a grant to make a research visit at EPCC, the Ministry of Education and Science and the former Ministry of Science and Innovation of Spain for the projects TIN2007-67537-C03-02, TIN2010-16735, TIN2010-12011-E and the FPU Grant AP2008-01578.

Jorge González Domínguez

Que ser valiente no salga tan caro,
que ser cobarde no valga la pena

Joaquín Sabina

Contents

List of Tables

List of Figures

List of Algorithms

Preface

The Partitioned Global Address Space (PGAS) programming model provides significant productivity advantages over traditional parallel programming paradigms. In the PGAS model all threads share a global address space, just as in the shared memory model. However, this space is logically partitioned among threads, just as in the distributed memory model. Thus, the data locality exploitation increases performance, whereas the shared memory space facilitates the development of parallel codes. As a consequence, the PGAS model has been gaining increasing attention. A number of PGAS languages are now ubiquitous, such as Titanium [109], Co-Array Fortran [22] and Unified Parallel C (UPC) [112].

UPC is an extension of ANSI C for parallel computing that follows the PGAS programming model and can potentially perform at similar levels to those of Message Passing Interface (MPI), scaling even up to thousands of processors with proper support from the compiler and the runtime system [37, 72]. In addition, the one-sided communications present in languages such as UPC have demonstrated they are able to obtain even better performance than the traditional two-sided communications [5, 7, 79, 80, 95, 96].

Currently, there are commercial and open source UPC compilers, such as Berkeley UPC [10], GCC UPC [43], Cray UPC or HP UPC [58], for nearly all parallel machines. However, a barrier to a more widespread adoption of UPC is the lack of parallel libraries for UPC developers.

The present PhD Thesis "UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations" focuses on developing a parallel numerical library that exploits the advantages of the PGAS programming model and, in particular, the UPC language. The distribution of these kinds of libraries can help to increase the acceptance of this paradigm by programmers of parallel numerical codes.

Currently, there is a trend towards developing codes that can automatically optimize their performance depending on the machine on which they are executed. Related to this, many optimization techniques have been developed for parallel computing, most of them being focused on improving the communication algorithms [38, 39, 98, 108, 127] or using the best mapping policies when assigning processes/threads to cores [13, 14, 19, 57, 74]. There are even approaches to optimize UPC programs according to the architecture of the system by using hybrid programming models [31, 117].

Among the different parallel architectures, clusters of multicores pose significant challenges, as they present a hybrid distributed and shared memory architecture with non-uniform communication latencies. Furthermore, because of the sharing of memory or even cache among cores, concurrent memory accesses using different threads or processes can decrease the overall memory access throughput.

In order to improve the performance of the UPCBLAS routines on these kinds of systems, this Thesis also tackles the development of Servet, a suite of benchmarks focused on detecting a set of parameters with high influence on the overall performance of multicore clusters. Servet is able to detect the cache hierarchy, including cache size and which caches are shared by each core, bandwidths and bottlenecks in memory accesses, as well as communication latencies among cores.

Besides performance, ease of use is a key factor in the adoption of the library by numerical application developers. For this reason programmability has been present in all the design decisions of the library.

Finally, all the lessons learned from the implementation of the dense routines were applied to the development of sparse counterparts.

The results of this research work have been published in: [44, 45, 46, 47, 48, 49, 50, 51, 52, 53] (references in Spanish not included). The Thesis is organized into six chapters and two appendices whose contents are summarized as follows:

- Chapter 1, *Background and State of the Art*, is intended to give the reader a clear background about parallel numerical libraries and PGAS languages so that the development of UPCBLAS can be completely understood. The chapter begins with an explanation of the characteristics that all the languages that follow the PGAS programming model share, a presentation of some examples of these kinds of languages and an exposition of their advantages and drawbacks. After that, it continues with a deeper explanation of the UPC language, including the memory model overview and some concepts necessary to understand the design, syntax and implementation of the UPCBLAS routines. The chapter finishes with a revision of the previously available parallel numerical libraries in the state of the art.

- Chapter 2, *Dense UPCBLAS*, presents all the stages of the development of UPCBLAS: design of the syntax and interface, implementation of the routines and inclusion of optimization techniques. The advantages related to programmability and productivity are also enumerated throughout the chapter.

- Chapter 3, *Servet: Measuring Hardware Parameters*, describes Servet, a benchmark suite that detects some hardware parameters very useful to implement architecture-aware optimization techniques to be included in UPCBLAS. The chapter begins presenting some background that proves the suitability of these kinds of techniques to optimize routines such as the UPCBLAS ones. Then, the benchmarks that determine the cache topology, the memory access bottlenecks and the communication layers among cores are described. Finally, the chapter explains the optimization techniques included in the UPCBLAS library that use Servet.

- Chapter 4, *Experimental Evaluation*, provides a performance evaluation of UPCBLAS. After the description of the two experimental testbeds (the Carver and HECToR supercomputers), execution times and speedups of six representative UPCBLAS routines are shown. Specifically, the dot, matrix-vector, outer and matrix-matrix products and the BLAS2 and BLAS3 triangular solvers. Furthermore, a comparison with MPI-based counterparts is also provided. The chapter also proves the utility of the library to implement more complex numerical libraries by explaining the development of linear solvers of equations through Cholesky and LU factorizations using UPCBLAS routines. The chapter concludes with the evaluation of the performance of these codes on one of the presented testbeds.

- Chapter 5, *Sparse Numerical Routines in UPC*, presents the implementation of the sparse dot, matrix-vector and matrix-matrix products in UPC. The goal of the chapter is to provide a first approach to implement sparse numerical codes using UPC, without sacrificing the productivity and programmability achieved in dense computation. The chapter discusses which aspects of the design and implementation of the dense routines included in UPCBLAS can be reused for sparse computation and which ones must be discarded because of the inherent irregular memory accesses to sparse vectors and matrices in these codes. The chapter also evaluates the suitability of six different sparse storage formats for UPC sparse numerical codes.

- Chapter 6, *Conclusions and Future Work*, summarizes the main contributions of the Thesis and outlines the main research lines that can be derived from this work.

- Appendix A, *UPCBLAS Interface*, shows the syntax of all the UPCBLAS routines, including a description of the parameters and some instructions about how to initialize the shared arrays before calling the routines.

- Appendix B, *Servet Library Interface*, enumerates the routines included in the Servet API, describing their syntax and functionality.

Chapter 1

Background and State of the Art

This chapter presents the background necessary to understand the development of UPCBLAS. On the one hand, it explains the main characteristics of the PGAS programming model and, particularly, UPC. On the other hand, the state of the art related to parallel numerical computations is also analyzed. Although the study of the related literature exposed that there were no parallel numerical libraries for any PGAS language before this Thesis, some good conclusions were obtained from previous works that developed numerical libraries for other parallel programming models or isolated routines for UPC.

The structure of this chapter is as follows: Section 1.1 presents the main characteristics, advantages and drawbacks of the PGAS programming model. The most important features of UPC that were taken into account to develop UPCBLAS are described in Section 1.2. Finally, Section 1.3 analyzes several numerical libraries built for other parallel programming models, focusing on determining their strengths and weaknesses. This last section also presents some research works that deal with numerical routines using PGAS languages.

1.1. PGAS Programming Model and Languages

There are two traditional programming models that have been used for years by parallel programmers. The most commonly used is the message-passing model, whose de facto standard is the Message Passing Interface (MPI) [75]. This model was initially designed for distributed memory machines, where each process has its own memory not shared by other processes. Programmers must explicitly distribute the data and perform the communications among processes, which is tedious and error-prone and thus decreases programmability. Furthermore, other major criticism of the message-passing model is that both sides have to agree on when messages are being sent and received and thus, typically, all communications have to be known ahead of time. The main strengths of MPI are its flexibility and portability, being able to achieve very good performance in very different systems.

The second traditional parallel approach is the shared memory programming model, where all threads can access the whole memory. The best known API is OpenMP [104], which is much easier to use than message-passing libraries as data are not logically distributed and can be directly accessed by threads without explicitly indicating communications.

In an attempt to improve performance on current architectures that mix shared and distributed memory, several works are based on hybrid MPI+OpenMP codes [64, 89]. However, these hybrid parallel codes are very difficult to program and are not portable, as they are very architecture-dependent and they need some tuning to obtain good performance on different machines. These problems could be overcome with the OpenMP runtime presented in [67] that achieves good performance on clusters, but it has not been proven on large supercomputers yet.

Nowadays, the best approach to increase the programmability in current clusters and supercomputers without compromising performance and portability is the usage of the PGAS paradigm [123], a parallel programming model that combines the advantages of the two traditional ones described above. Figure 1.1 shows the memory model of the PGAS languages where there is a global shared address space exposed to the user which is logically divided among threads, so each thread is associated or presents affinity to a part of the shared memory. The PGAS languages explicitly expose the non-uniform nature of memory access times: operations on local data (i.e. the portion of the address space that a particular processor has affinity to) will be much faster than operations on remote data (i.e. any part of the shared address space that a processor does not have affinity to). The knowledge of the strengths of the PGAS languages has even led some researchers to try to include shared arrays

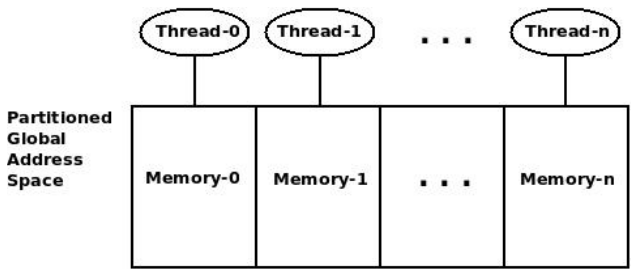


Figure 1.1: Memory model in PGAS languages

into MPI codes [30]. The main advantages of the PGAS programming model are:

- The global shared address space facilitates the development of parallel codes, allowing all threads to directly read and write remote data and avoiding the error-prone data movements of the message-passing paradigm.
- The accesses to shared memory also allow to develop efficient one-sided communications that can outperform the traditional two-sided ones, as a thread can directly read and write on remote memory without the explicit cooperation of the thread on the remote core [7, 24].
- Compared to the shared memory paradigm, the performance of the codes can be increased by taking into account data affinity as typically the accesses to remote data will be much more expensive than the accesses to local data.
- The PGAS languages provide a programming model that can be used across the whole system instead of relying on two distinct programming models that must be melded together, as in the hybrid OpenMP+MPI solution. Thus the PGAS languages aim to deliver the same performance as hybrid approaches but using a uniform programming model.

There are several PGAS languages that are gaining attention apart from UPC, which is the most commonly used and will be further explained in Section 1.2:

- Co-Array Fortran (CAF) [22]: It was created as a small set of extensions that, being based on a static collection of asynchronous process images, introduces coarrays to Fortran 95 in order to support the PGAS programming model.

It is included in the Fortran 2008 standard and during the last years several research works were focused on improving and extending it [73, 82, 92]. Its good implementation of the one-sided communications allows Co-Array Fortran to obtain similar or even better performance than MPI depending on the architecture [23, 24, 25, 63].

- Titanium [109]: It is essentially a superset of Java 1.4 that provides a global memory space abstraction but maintaining all the expressiveness, usability and safety properties of that object oriented language. Current compilers translate Titanium programs entirely into C, where they are compiled to native binaries by a C compiler and then linked to the Titanium runtime libraries. Thus, no Java Virtual Machine is necessary. The main advantage of this language is its high programmability, as it inherits the programmability of Java [118, 119]. Besides, its performance is not far from MPI for the NAS Parallel Benchmarks [29].

- Chapel (Cascade High Productivity Language) [102]: It is a parallel programming language developed by Cray as part of the Cray Cascade project, a participant in DARPA's High Productivity Computing Systems (HPCS) [56]. Chapel was designed from scratch rather than by extending an existing language. It is an imperative block-structured language, designed to reduce the gap to learn parallel programming for users of sequential C, C++, Fortran, Java, Perl or Matlab [18]. Chapel specifies a *locale* type that enables users to choose the placement of data and tasks on a target architecture in order to optimize the code thanks to locality. Moreover, Chapel supports global-view data sets (called *aggregates*) with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. The main strength of Chapel is that it provides a spectrum of features at various levels so that code which is less performance-oriented can be written more easily. However, if additional performance is required for a section of code, programmers are able to rewrite it using low-level performance-oriented features until they are able to obtain their target performance, not only on clusters but also on GPUs [97].

- X10 [122]: It is another parallel programming language related to the HPCS project, in this case built by IBM as part of the Productive, Easy-to-use, Reliable Computing System (PERCS) project. Each computation is divided among a set of *places*, each of which holds some data and hosts one or more *activities* that operate on those data. One of its most interesting novelties is that X10 uses the concept of parent and child relationships for activities to prevent the lock stalemate that can occur when two or more processes wait for each other to finish before they can complete. It also provides a rich array sublanguage. The community of X10 users is quite active, taking advantage of its high programmability [76] and increasing its performance with novel optimizations [3, 91].

- Fortress [87]: This is the third language that started being funded by the HPCS project. However, Sun was dropped from the HPCS project in 2006 and Fortress was transformed into an open-source project with an open-source community. Syntactically, it is similar to Scala or Haskell, emulating mathematical notation as closely as possible.

1.2. Overview of Unified Parallel C

This section describes the most important features of the memory model in UPC as they were taken into account to design the interface of the library, determine the most appropriate data distributions among threads and implement the numerical routines. An overview of the different types of pointers present in the language and the mechanisms to access remote data are also included as their behavior is the basis for some of the optimization techniques included in UPCBLAS.

As mentioned previously, all PGAS languages, and thus UPC, expose a global shared address space to the user which is logically divided among threads, so each thread is associated or presents affinity to a part of the shared memory. Moreover, UPC also provides a private memory space per thread for local computations, as shown in Figure 1.2. Therefore, each thread has access to both its private memory and to the whole global space, even the parts that do not present affinity to it. This memory specification combines the advantages of both the shared and distributed programming models. On the one hand, the global shared memory space facilitates the development of parallel codes, allowing all threads to directly read and write

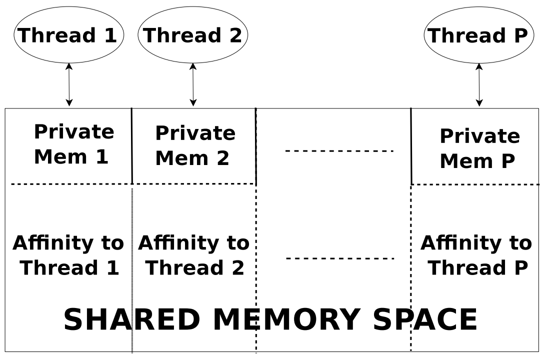


Figure 1.2: Memory model in UPC

remote data without explicitly notifying the owner. On the other hand, the performance of the codes can be increased by taking into account data affinity. Typically the accesses to remote data will be much more expensive than the accesses to local data (i.e. accesses to private memory and to shared memory with affinity to the thread).

Shared arrays are employed to implicitly distribute data among all threads, as shared arrays are spread across the threads. The syntax to declare a shared array `A` is: `shared [BLOCK_FACTOR] type A[N]`, `BLOCK_FACTOR` being the number of consecutive elements with affinity to the same thread, `type` the datatype, and `N` the array size. This means that the first `BLOCK_FACTOR` elements are associated to thread 0, the next `BLOCK_FACTOR` to thread 1, and so on. Thus, the element `i` in the array has affinity to the thread $\lfloor \frac{i}{BLOCK\_FACTOR} \rfloor mod(THREADS)$, $THREADS$ being the total number of threads in the UPC execution. There are two additional ways to declare certain shared arrays:

- `shared type A[N]` specifies a cyclic distribution (`BLOCK_FACTOR = 1`).
- `shared [] type A[N]` declares an array with indefinite `BLOCK_FACTOR`, which means that all the elements are stored in the shared memory with affinity to thread 0 (`BLOCK_FACTOR = N`).

As an extension of the C language, UPC provides functionality to access memory through pointers. Due to the two types of memory available in the language, several types of pointers arise:

- Private pointers (*from private to private*). They are only available for the thread that stores them in its private memory and can reference addresses in the same private memory or in the part of the shared memory with affinity to the owner. Their syntax is the same of standard C pointers: `type *p`

- Private pointers to shared memory (*from private to shared*). They are only available for the thread that stores them in its private memory, but can have access to any data in the shared space. They contain three fields in order to know their exact position in the shared space: the *thread* where the data is located, the *block* that contains the data and the *phase* (the location of the data within the block). Thus, when performing pointer arithmetic on a pointer-to-shared all three fields will be updated, making the operation slower than private pointer arithmetic. As in shared arrays, the `BLOCK_FACTOR` can be specified: `shared [BLOCK_FACTOR] type *p`

- Shared pointers (*from shared to shared*). They are stored in shared memory (and therefore accessible by all threads) and they can access any data in the shared memory. Their complexity is similar to private pointers to shared memory. They are defined as: `shared [BLOCK_FACTOR] type *shared p`

- Shared pointers to private memory (*from shared to private*). They are stored in shared memory and point to the private space. However, their use is not advisable and they are not available in some compiler implementations.

The most intuitive way for one thread to access remote data is using pointers to shared memory. Nevertheless, UPC also provides functions to move blocks of data between two parts of the memory:

- `upc_memget`: One thread copies one block of data from shared memory (with or without affinity) to its private memory.

- `upc_memput`: One thread copies one block of data from its private memory to shared memory (with or without affinity).

- `upc_memcpy`: One thread copies one block of data from any shared memory position to any other shared memory position.

- `upc_memset`: Assigns the same value to all the elements of a block of data in shared memory.

The usage of these routines allows UPC programmers to aggregate remote data copies and thus obtain better performance than using several single data copies through pointers to shared. Some compilers also provide versions of these functions that perform asynchronous copies to overlap communication and computation [12]. The return of each of these functions only indicates that the copy has started and a subsequent synchronization on the completion of that operation is required before guaranteeing that all data were copied. Although these asynchronous copies are not included in the standard reference of the language, they are included in the most important UPC compilers (Berkeley UPC, Cray UPC, HP UPC).

1.3. Parallel Numerical Computation in the Literature

Numerical libraries have been used for decades in order to help programmers to develop their numerical codes easily and with high performance. UPCBLAS provides a relevant subset of the *Basic Linear Algebra Subprograms* (BLAS) routines [6, 33] implemented for UPC. The BLAS interface was first published in 1979 and it has become a standard as it is widely used by scientists and engineers. The interface was initially developed for Fortran but now there is also a C interface [11]. BLAS routines are divided into three levels:

- BLAS 1 level: Scalar-vector and vector-vector operations.
- BLAS 2 level: Matrix-vector operations.
- BLAS 3 level: Matrix-matrix operations.

Although BLAS initially focused on dense and banded operations, nowadays many applications use sparse vectors and matrices. Thus, currently the BLAS library includes the SparseBLAS one [36, 99], which provides computational routines for unstructured sparse matrices, that is, matrices that do not present a particular sparsity pattern. Another well-known interface for numerical computation is the Linear Algebra Package (LAPACK) [69], that internally uses the BLAS library to implement more complex numerical routines such as linear solvers or matrix factorizations. Many vendors have developed their own numerical libraries, such as Intel MKL [62], AMD ACML [2], Cray LibSci, IBM ESSL [103] or HP MLIB [59], that include, among others, their own implementations of the BLAS, SparseBLAS and LAPACK routines very optimized for their architectures. Furthermore, there are in the literature several numerical libraries with support for parallel dense matrix computations based on the message-passing model. Among them, PBLAS [21, 83], a subset of BLAS, and ScaLAPACK [105], a subset of LAPACK, are the most popular. Based on them, Aliaga et al. [1] made an effort to parallelize the open source numerical library GSL [54].

The main drawback of the message-passing-based libraries is that they need an explicit data distribution that increases the effort required to use them. Users are forced to deal with specific structures defined in the library and to work in each process only with the part of the matrices and vectors stored in the local memory. Therefore, users must be aware of the appropriate local indices to use in each process, which increases the complexity of developing parallel codes [68]. In the literature there exist some proposals that try to ease the use of message-passing numerical libraries, such as PyScaLAPACK [34] and Elemental [86]. Following this trend, one of the goals of UPCBLAS is to increase programmability. The PGAS languages in general, and UPC in particular, offer productivity advantages compared to the message-passing model. In [16] the number of lines of code needed by the MPI and the UPC implementations of the NAS Parallel Benchmarks and other kernels are compared. Similar statistical studies with university students are presented in [84] and [101]. These works have demonstrated that the effort needed to solve the same problem is lower in UPC than in MPI. Furthermore, the global address space in UPC allows hiding the complex local index generation for matrices and vectors as well as data movement issues present in the message-passing approaches. The experimental evaluation of the UPCBLAS library will show that simplicity does not significantly impact performance.

Regarding other proposals of PGAS libraries, a parallel numerical library that combines the object-oriented-like features of Fortran 95 with the parallel syntax of Co-Array Fortran was presented in [81]. However, its object-oriented layer leads to use object maps, additional structures to work with distributed matrices and vectors similarly to the message-passing-based libraries, which increases the effort needed to parallelize sequential numerical algorithms. Travinin and Kepner [111] developed pMatlab, a parallel library built on top of MatlabMPI (a set of Matlab scripts that implement a subset of MPI). It works with matrices and vectors distributed by simulating a pure PGAS scenario in order to take advantage of the ease of programming and a higher level of abstraction.

Focusing on numerical computations in UPC, Bell and Nishtala present in [8] a sparse triangular solver in UPC. In [61] Husbands and Yelick undertake the parallelization of the LU factorization. However, these works do not take advantage of the ease of use of the global shared memory in UPC as the matrices and vectors are initially distributed in the private memory of the threads, in the same way as in message-passing numerical libraries.

Chapter 2

Dense UPCBLAS

This chapter explains the whole development of the dense routines included in UPCBLAS. It is divided into two parts: Section 2.1 describes in detail all the decisions taken during the design of the different levels of UPCBLAS. It also includes some conclusions that summarize the programmability and productivity advantages of UPCBLAS compared to other parallel numerical routines. Section 2.2 enumerates several optimizations included within the routines to improve their performance.

2.1. UPCBLAS Design

One of the main reasons to create a library is to facilitate the development of new codes for the target language. Thus, the design stage must be carried out carefully in order to provide functions with an interface easy to use by programmers.

Programmability has been an important factor in all the design decisions of the library. Specifically, UPCBLAS uses shared arrays to represent distributed matrices and vectors, which are implicitly partitioned among threads. Thus, the complex steps of declaring and distributing vectors and matrices required by the MPI-based libraries are avoided.

In general, programs that use parallel numerical libraries must carry out the following steps: 1) create the structures to represent the distributed vectors and matrices; 2) distribute the data of the vectors and matrices into these structures;

3) call the numerical functions using the structures as parameters; 4) perform other operations with the distributed data (e.g. gathering or reducing some elements so that they are in the local memory of one process, write some elements of all or some processes in a file...); 5) release the structures.

The message-passing paradigm (e.g. MPI) does not provide any structure in the language to deal with vectors and matrices distributed among the processes. Hence, developers of message-passing numerical libraries have to create additional structures to represent distributed vectors and matrices. Both the new structures and the 2D distribution of the matrices are concepts that pose an important challenge for most of the users of parallel numerical libraries (researchers and engineers from different areas), as can be seen in the results of the survey [68]. In contrast, UPC libraries can make use of shared arrays, making steps 1, 2, 4 and 5 almost trivial. Therefore, the design of UPCBLAS, based on these shared arrays, significantly improves the ease of use of the library and thus the productivity of numerical applications developers. Furthermore, UPCBLAS also facilitates the third step by simplifying the interface of the routines because the syntax of the UPCBLAS functions is quite similar to the syntax of the corresponding sequential BLAS routines (see Appendix A). They only change the type of the pointers (so they can point to shared memory) and include additional parameters to indicate the type of data distribution.

However, the use of UPC shared arrays to distribute vectors and matrices imposes some limitations on the types of distributions that can be performed. On the one hand, the block factor (see Section 1.2) must be constant for all the threads. On the other hand, multidimensional distributions are not allowed. Thus, for some UPCBLAS routines, the distribution that theoretically obtains the best performance (e.g. 2D distributions for the matrix-matrix product) is not available. Nevertheless, the interface is flexible enough so that, if multidimensional distributions were allowed in UPC shared arrays in the future, no changes would be necessary to take advantage of the 2D distributions. Even more novel distributions like the 2.5D algorithms (with or without overlapping of communications and numerical computations) developed for the BLAS3 routines in [44] could be included.

Table 2.1 lists all the implemented routines, a representative subset of BLAS. A total of 52 different functions were implemented: 13 routines and 4 datatypes per routine. Next subsections show the interface and main characteristics of the

Table 2.1: UPCBLAS routines. All the functions follow the naming convention: `upc_blas_Tblasname`, where "T" represents the data type (s=float; d=double; c=single precision complex; z=double precision complex) and `blasname` is the name of the routine in the sequential BLAS library

BLAS level	Tblasname	Action
BLAS1	Tcopy	Copies a vector
	Tswap	Swaps the elements of two vectors
	Tscal	Scales a vector by a scalar
	Taxpy	Updates a vector using another one: $y = \alpha * x + y$
	Tdot	Dot product between two vectors
	Tnrm2	Euclidean norm of a vector
	Tasum	Sums the absolute value of the elements of a vector
BLAS2	Tgemv	Matrix-vector product
	Tger	Outer product between two vectors
	Ttrsv	Solves a triangular system of equations
BLAS3	Tgemm	Matrix-matrix product
	Ttrsm	Solves a block of triangular systems of equations
	Tsyrk	Product of a symmetric matrix by its transpose

UPCBLAS routines. The exact syntax of all the UPCBLAS routines is detailed in Appendix A.

All the routines return a local integer error value which refers only to each thread execution. In order to ensure that no error has occurred in any thread, the global error checking must be made by the programmer using the local error values. This is a usual practice in parallel libraries to avoid unnecessary synchronization overheads.

2.1.1. BLAS1 Routines

In order to favor the adoption of UPCBLAS among PGAS programmers, the syntax of these functions is similar to the standard collectives library [112]. For instance, the syntax of the single precision dot product is:

```
int upc_blas_sdot(int block_size, int size, shared void *x,
                  shared void *y, shared float *dst);
```

`x` and `y` being the source vectors of length `size`; `dst` the pointer to shared memory where the dot product result will be written; and `block_size` the block factor (see `BLOCK_FACTOR` in Section 1.2) of the source vectors. For performance reasons, the block factor must be the same for both vectors. This function treats pointers `x` and `y` as if they had type `shared [block_size] float[size]`.

An important design decision in UPCBLAS is that, looking for efficiency, only one `block_size` parameter to indicate the same block factor for both shared arrays is included. Figure 2.1 shows two scenarios for the dot product. When both vectors have the same `block_size`, all the pairs of elements that must be multiplied are stored in the shared memory with affinity to the same thread. Thus, each thread only has to perform its partial dot product in a sequential way and the final result is obtained through a reduction operation over all threads. However, in the second case, several remote accesses that affect performance are necessary so that each thread can obtain all the data needed in its partial dot product.

Finally, in order to be able to work with subvectors, `x` and `y` do not need to point to the first element of a shared array. Figure 2.2 illustrates an example for a subvector that is stored from position 2 to 13 of a shared array. The only restriction is that the subvector must start in the first position of a block (i.e. its phase must be 0). This restriction is not a big issue as it is also present in the standard UPC libraries (e.g. the collectives library) and it is the natural way to declare and allocate shared arrays.

2.1.2. BLAS2 Routines

Shared matrices in UPC can only be distributed in one dimension as the UPC syntax does not allow multidimensional layouts. The definition of multidimensional block factors has been proposed in [4], although currently this extension is not included in the language specification. Therefore, all the UPCBLAS routines rely on the 1D data distribution present in the standard. An additional parameter

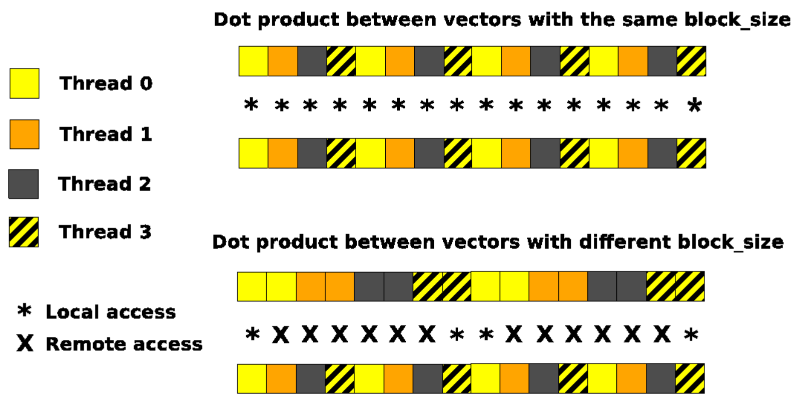

Dot product between vectors with the same block_size
Thread 0
Thread 1
Thread 2
Thread 3
Dot product between vectors with different block_size
* Local access
X Remote access
* * * * * * * * * * * * * * * *
* X X X X X X * * X X X X X X *

Figure 2.1: Remote and local accesses in `upc_blas_sdot` according to the block factor of the source vectors

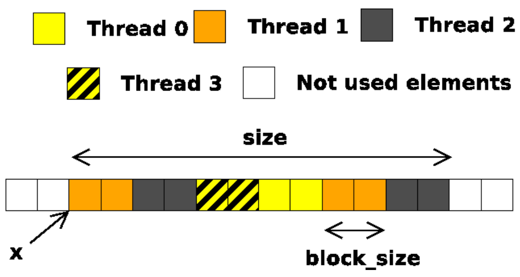


Figure 2.2: Meaning of the parameters of `upc_blas_sdot` when working with sub-vectors

(`dimmDist`) is needed in the routines to indicate the dimension used for the distribution of the matrix. For instance, the UPCBLAS routine for the single precision matrix-vector product ($y = \alpha * A * x + \beta * y$) is:

```
int upc_blas_sgemv(UPCBLAS_DIMMDIST dimmDist, int block_size,
                  int sec_block_size, UPCBLAS_TRANSPOSE transpose,
                  int m, int n, float alpha, shared void *A,
                  int lda, shared void *x, float beta,
                  shared void *y);
```

`A` and `x` being the source matrix and vector, respectively; `y` the result vector; `transpose` an enumerated value to indicate whether matrix `A` is transposed; `m` and `n` the number of rows and columns of the matrix; `alpha` and `beta` the scale factors for `A` and `y`, respectively; and `dimmDist` another enumerated value to indicate if the source matrix is distributed by rows or columns. `lda` is a parameter inherited from the sequential BLAS library to work with submatrices. It expresses the memory distance between two elements in the same column and in consecutive rows of the submatrix.

Figure 2.3 shows an example with the row and column distributions when `A` is a submatrix that discards two rows and two columns of the global array. Thanks to using arrays stored in shared memory and pointing directly to the first element of the submatrix, the `lda` parameter is enough to specify all the information to work with submatrices, as in sequential BLAS routines. Therefore, the syntax of the routines is simpler than in message-passing-based numerical libraries where the first row and column of the submatrix must be explicitly specified through additional parameters. Similarly to the UPC BLAS1 routines, the only restriction is that the submatrix must start at the first row/column of a block in the row/column distribution. This approach is followed in all the UPC BLAS2 and BLAS3 routines to work with submatrices.

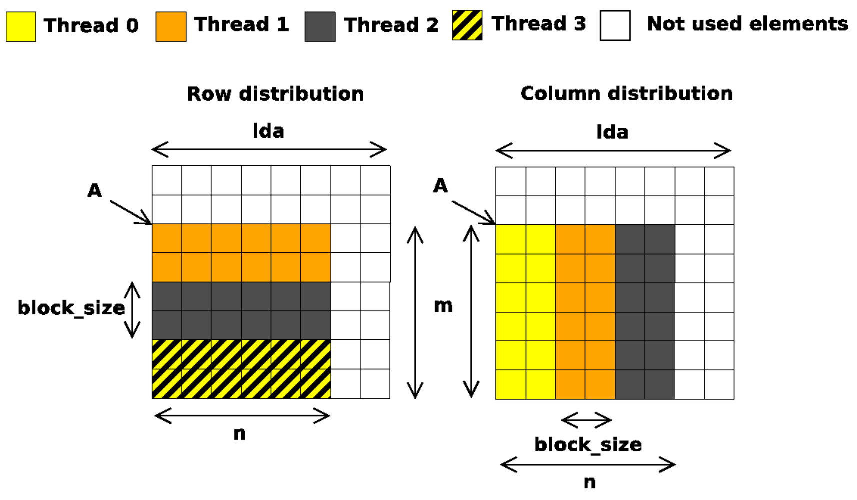


Figure 2.3: Meaning of the parameters of `upc_blas_sgemv` when working with a submatrix

The meaning of the `block_size` and `sec_block_size` parameters depends on the `dimmDist` value:

- If the source matrix `A` is distributed by rows (`dimmDist=upcblas_rowDist`), `block_size` is the number of consecutive rows with affinity to the same thread and `sec_block_size` the block factor related to the source vector `x`. For instance, in the non-transpose case, this function treats pointers as:

- `A`: `shared[block_size*lda] float[m*lda]`
- `x`: `shared[sec_block_size] float[n]`
- `y`: `shared[block_size] float[m]`

▪ If the source matrix is distributed by columns (`dimmDist=upcblas_colDist`), `block_size` is the number of consecutive columns with affinity to the same thread and `sec_block_size` the block factor related to the result vector `y`:

- `A`: `shared[block_size] float[m*lda]`
- `x`: `shared[block_size] float[n]`
- `y`: `shared[sec_block_size] float[m]`

Figure 2.4 illustrates the behavior of the matrix-vector product when `A` is distributed by rows (in this example `block_size`=2). In order to exploit data locality as much as possible each thread only accesses the rows of the matrix with affinity to that thread. Then, by applying a sequential partial matrix-vector product with these rows and all the elements of `x`, each thread calculates a partial result that corresponds with its rows of `A`. Thus, if the distribution of the result vector matches the distribution of the matrix, all the partial results can be copied to their correct final positions working only with local memory. This is the reason why in the row case the parameter `block_size` indicates not only the distribution of the matrix, but also the distribution of the result vector. Thus, users are forced to declare `y` with a block factor equal to `block_size` in order to guarantee always a good performance. As all the elements of `x` must be used by all threads, its block factor does not need to be linked to the distribution of the matrix, and it is indicated through the `sec_block_size` parameter.

Figure 2.5 shows the behavior of the routine with a column distribution. In this case the source vector `x` must have always the same distribution (`block_size`) as the matrix and the distribution of the result vector `y` is passed through the `sec_block_size` parameter. In order to compute the i^{th} element of the result, the

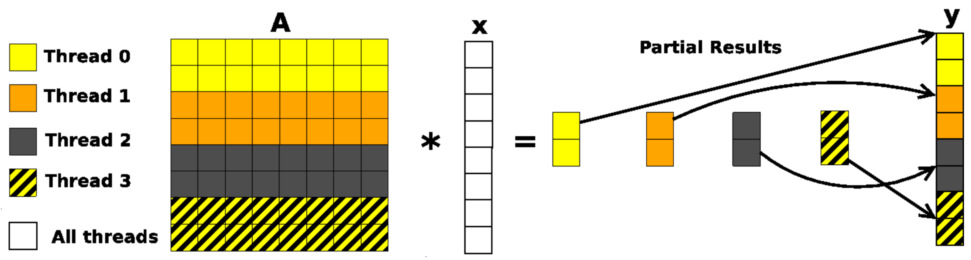


Figure 2.4: Matrix-vector product (*Tgemv*) using a row distribution for the matrix

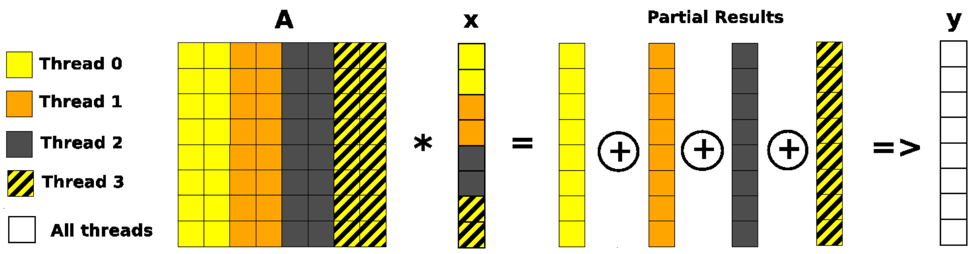


Figure 2.5: Matrix-vector product (*Tgemv*) using a column distribution for the matrix

i^{th} values of all partial results must be added. These additions need reduction operations involving all UPC threads, so their performance is usually poor.

The approach to parallelize the outer product (*Tger*) within UPCBLAS is quite similar to the matrix-vector product. The main difference is that the parameter `dimmDist` is related to the result matrix ($A = \alpha * x * y^T + A$). The syntax for single precision is:

```
int upc_blas_sger(UPCBLAS_DIMMDIST dimmDist, int block_size,
                  int sec_block_size, int m, int n, float alpha,
                  shared void *x, shared void *y, shared void *A,
                  int lda);
```

`x` and `y` being the source vectors; `A` the result matrix; `m` and `n` the number of rows and columns of the matrix; `alpha` the scale factor; and `dimmDist` an enumerated value to indicate if the matrix is distributed by rows or columns.

Again, the exact meaning of the `block_size` and `sec_block_size` parameters depends on the `dimmDist` value:

- If matrix `A` is distributed by rows (`dimmDist=upcblas_rowDist`), the data distribution is as specified in Figure 2.6. In this case each thread only needs to perform an outer product with the elements of `x` that correspond to the row distribution of `A` and the whole vector `y`. Therefore, pointers must be:
  - `x`: `shared[block_size] float[m]`
  - `y`: `shared[sec_block_size] float[n]`
  - `A`: `shared[block_size*lda] float[m*lda]`
- If the matrix is distributed by columns, `block_size` is also related to vector `y` and `sec_block_size` is the block factor related to vector `x`, as all threads need to have access to all its elements (see Figure 2.7):
  - `x`: `shared[sec_block_size] float[m]`
  - `y`: `shared[block_size] float[n]`
  - `A`: `shared[block_size] float[m*lda]`

The routine to solve a triangular system of equations $M * x = b$ (*Ttrsv*) is a special case among the UPC BLAS2 routines because there are a lot of data dependencies in the internal algorithm. In the BLAS interface vector `b` is always overwritten by the solution vector `x`, so both are represented by the same parameter. According to this assumption, the syntax of the UPC BLAS2 triangular solver for single precision is:

```
upc_blas_strsv(UPCBLAS_DIMMDIST dimmDist, int block_size,
               UPCBLAS_UPLO uplo, UPCBLAS_TRANSPOSE transpose,
               UPCBLAS_DIAG diag, int n, shared void *M, int ldm,
               shared void *x);
```

`nxn` being the size of the triangular matrix `M` and `n` the length of `x`. The enumerated values `uplo`, `transpose` and `diag` are included to determine the characteristics of `M` (upper/lower triangular, transpose/non-transpose, elements of the diagonal equal to one or not). In this routine, all the distributions are specified by `block_size` and the vector and the matrix must be stored in shared arrays with the following syntax:

- M: `shared[block_size*ldm] float[n*ldm]` if row distribution
- M: `shared[block_size] float[n*ldm]` if column distribution
- x: `shared[block_size] float[n]` in both cases

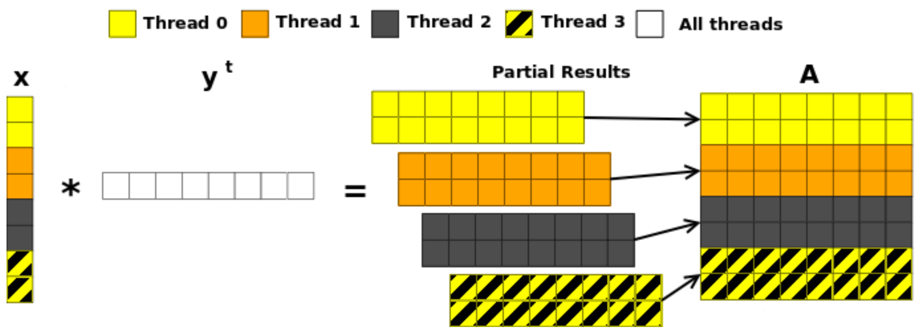


Figure 2.6: Outer product (*Tger*) using a row distribution for the matrix

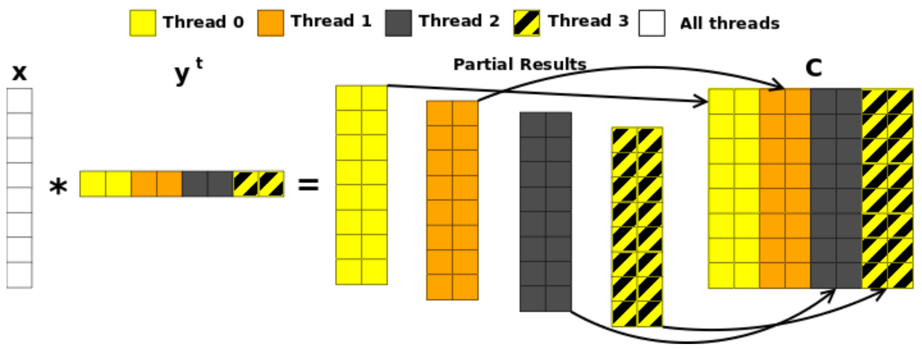


Figure 2.7: Outer product (*Tger*) using a column distribution for the matrix

Figure 2.8 shows an example of the distribution by rows of a lower triangular coefficient matrix using two threads and two blocks per thread. The triangular matrix is logically divided into square blocks M_{ij}. These blocks are triangular submatrices if $i = j$, square submatrices if $i > j$, and null submatrices if $i < j$. Algorithm 2.1 shows the parallel algorithm for this example. Once one thread computes its part of the solution (output of the sequential `trsv` routine), it is broadcast to all threads so that they can update their local parts of b with the sequential product (`gemv`). Thanks to specifying the distribution of both `M` and `x` with the same parameter

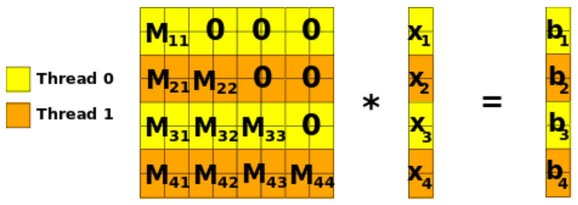


Figure 2.8: Matrix distribution for the parallel BLAS2 triangular solver (*Ttrsv*)

Algorithm 2.1 Algorithm for the BLAS2 triangular solver (*Ttrsv*) distributed by rows

```
$x_1 \leftarrow Solve\ M_{11} * x_1 = b_1$        BLAS Ttrsv()
$Broadcast\ x_1$
$b_2 \leftarrow b_2 - M_{21} * x_1$              BLAS Tgemv()
$b_3 \leftarrow b_3 - M_{31} * x_1$              BLAS Tgemv()
$b_4 \leftarrow b_4 - M_{41} * x_1$              BLAS Tgemv()
$x_2 \leftarrow Solve\ M_{22} * x_2 = b_2$        BLAS Ttrsv()
$Broadcast\ x_2$
$b_3 \leftarrow b_3 - M_{32} * x_2$              BLAS Tgemv()
...
```

(`block_size`), all sequential `trsv` and `gemv` computations can be performed without any communication except the broadcast. Note that all operations between two synchronizations (broadcasts) can be performed in parallel.

The column distribution would involve a nearly sequential algorithm with poor performance due to the characteristics of its dependencies. However, it is also available in the *Ttrsv* routine to allow a distribution reuse just in case the source matrix uses that distribution in other UPCBLAS routines within the same application.

2.1.3. BLAS3 Routines

In the BLAS3 routines, as there is more than one matrix, the number of possible combinations of distributions of the matrices grows. In the design of UPCBLAS programmability is a must and thus, in order to simplify the understanding and use of the UPC BLAS3 routines, the parameter `dimmDist` always makes reference to the result matrix. Moreover, this choice allows reusing the output data as input matrix in consecutive calls to UPCBLAS routines.

The interface of the UPCBLAS routine for a single precision matrix-matrix product ($C = \alpha * A * B + \beta * C$) is:

```
upc_blas_sgemm(UPCBLAS_DIMMDIST dimmDist, int block_size,
               int sec_block_size, UPCBLAS_TRANSPOSE transposeA,
               UPCBLAS_TRANSPOSE transposeB, int m, int n, int k,
               float alpha, shared void *A, int lda,
               shared void *B, int ldb, float beta, shared void *C,
               int ldc);
```

`mxk`, `kxn` and `mxn` being the sizes of `A`, `B` and `C`, respectively. `block_size` means the number of consecutive rows or columns of `C` (depending on the `dimmDist` value) with affinity to the same thread. In addition, `sec_block_size` is related to `B` in the row distribution and to `A` in the column one. The meaning of the rest of parameters is similar to the *sgemv* routine explained in the previous section. Thus, UPCBLAS *sgemm* with the row distribution treats pointers as:

- `A`: `shared[block_size*lda] float[m*lda]`
- `B`: `shared[sec_block_size] float[k*ldb]`
- `C`: `shared[block_size*ldc] float[m*ldc]`

If the column distribution is used, the matrices must be stored in arrays declared as follows:

- `A`: `shared[sec_block_size] float[m*lda]`
- `B`: `shared[block_size] float[k*ldb]`
- `C`: `shared[block_size] float[m*ldc]`

Figure 2.9 shows an example for the row distribution of the matrix-matrix product. As `block_size` is related to the result matrix `C`, in order to perform its sequential partial matrix-matrix product each thread needs to access the same rows of `A` than those of `C` with affinity to that thread, and all the elements of `B`. So, as in the equivalent distribution of the BLAS2 routine (*Tgemv*), `block_size` is related to `C` and `A`, and the distribution of `B` is determined by `sec_block_size`.

Figure 2.10 shows the same example when matrix `C` is distributed by columns. In order to perform the partial sequential matrix-matrix product each thread needs the whole matrix `A` but only the same columns of `B` as those of `C` with affinity to that thread. Thus, `block_size` also defines the distribution of `B` and `sec_block_size` is related to `A`. Unlike the column distribution of the BLAS2 routine (see Figure 2.5), no reductions are necessary in this case, avoiding the associated overhead at the end of the routine.

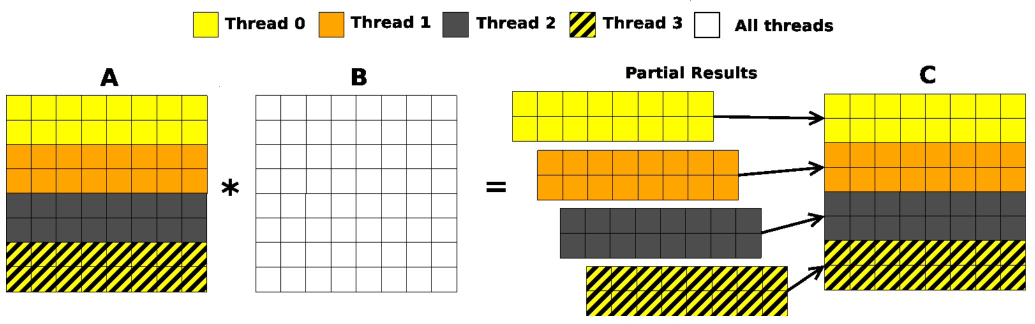


Figure 2.9: Matrix-matrix product (*Tgemm*) using a row distribution for matrix `C`

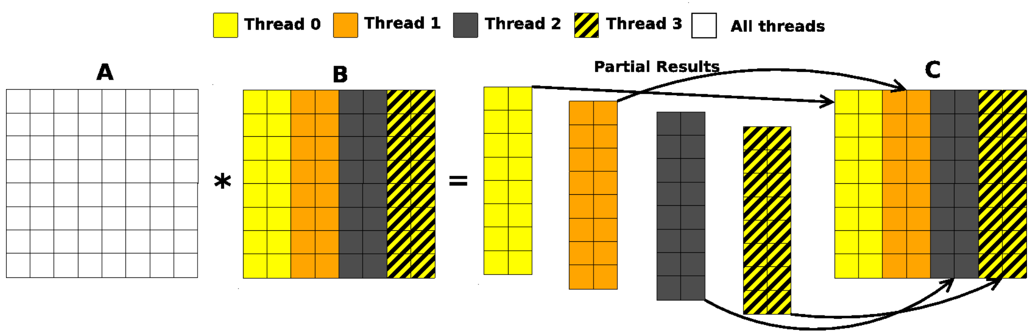


Figure 2.10: Matrix-matrix product (*Tgemm*) using a column distribution for matrix `C`

Regarding the BLAS3 triangular solver ($M * X = \alpha * B$, with matrix `B` overwritten by the result matrix `X`), the parameters are quite similar to those of the BLAS2 counterpart (*Ttrsv*), only changing the vectors by matrices and adding a new enumerated parameter (`side`) to indicate if the triangular matrix `M` is in the left or in the right part of the operation. `M` is `mxm` if it is left-sided or `nxn` if right-sided, and `X` is always `mxn`. The syntax for single precision is:

```
upc_blas_strsm(UPCBLAS_DIMMDIST dimmDist, int block_size,
               int sec_block_size, UPCBLAS_SIDE side,
               UPCBLAS_UPLO uplo, UPCBLAS_TRANSPOSE transpose,
               UPCBLAS_DIAG diag, int m, int n, float alpha,
               shared void *M, int ldm, shared void *X, int ldx);
```

Algorithm 2.2 Algorithm for the BLAS3 triangular solver (*Ttrsm*) distributed by rows

```
$X_1 \leftarrow Solve\ M_{11} * X_1 = B_1$                BLAS Ttrsm()
$Broadcast\ X_1$
$B_2 \leftarrow B_2 - M_{21} * X_1$                       BLAS Tgemm()
$B_3 \leftarrow B_3 - M_{31} * X_1$                       BLAS Tgemm()
$B_4 \leftarrow B_4 - M_{41} * X_1$                       BLAS Tgemm()
$X_2 \leftarrow Solve\ M_{22} * X_2 = B_2$                BLAS Ttrsm()
$Broadcast\ X_2$
$B_3 \leftarrow B_3 - M_{32} * X_2$                       BLAS Tgemm()
...
```

As in the matrix-matrix product, the distribution specified by `dimmDist` and `block_size` is always related to the result matrix. In this case the choice between row or column distribution leads to apply a different algorithm. If the result matrix `X` is distributed by rows, the routine performs this solver using the Algorithm 2.2, which is similar to Algorithm 2.1, but replacing the sequential BLAS2 routines *Tgemv* and *Ttrsv* by their equivalent BLAS3 *Tgemm* and *Ttrsm*, respectively. Thus, the triangular matrix is forced to have the same distribution as the resultant one:

- `M`: `shared[block_size*ldm] float[m*ldm]`
- `X`: `shared[block_size*ldx] float[m*ldx]`

However, if the result matrix `X` is distributed by columns, a similar approach to the column distribution of the matrix-matrix product, with independent sequential computations, is applied. This approach treats pointers as follows:

- `M`: `shared[sec_block_size] float[m*ldm]`
- `X`: `shared[block_size] float[m*ldx]`

Figure 2.11 shows an example of this column distribution. As the source matrix `B` is overwritten by the result matrix `X`, they are represented by the same pointer and thus they have the same block factor. Therefore, each thread has access to all the elements of the triangular matrix and applies a sequential *Ttrsm* routine to the columns of `B` and `X` with affinity to it. This approach improves performance by avoiding the dependencies present in the row distribution.

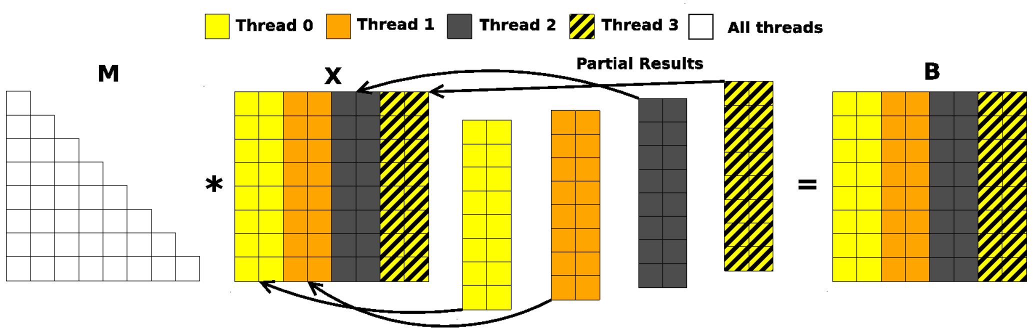


Figure 2.11: BLAS3 triangular solver (*Ttrsm*) using a column distribution

2.2. UPCBLAS Implementation

Besides improving productivity, parallel libraries are expected to offer good performance so users can take advantage of them to improve their applications. Although UPCBLAS has the limitation of working only with 1D distributions, a set of optimization techniques have been applied in the implementation of the routines to achieve the best possible performance:

- Data locality optimization: As explained in Section 2.1, the data distributions required to users when calling the UPCBLAS routines are focused on minimizing the remote accesses and, thus, decreasing the overhead due to communication among cores.

- Space privatization: As explained in Section 1.2, working with shared pointers is slower than with private ones. Experimental measurements in [37] and [128] have shown that the use of shared pointers increases execution times by up to several orders of magnitude. Thus, in our routines, when dealing with shared data with affinity to the local thread, the access is performed through standard C pointers instead of using UPC pointers to shared memory.

- Aggregation of remote shared memory accesses: Instead of the costly one-by-one accesses to remote elements, our routines perform remote shared memory accesses through bulk copies, using the `upc_memget()` and `upc_memput()` functions on remote bulks of data required by a thread. For instance, the vector `x` in Figure 2.4 is replicated in all threads using bulk copies of `sec_block_size` elements.

- Use of phaseless pointers: Many UPC compilers (including Berkeley UPC [20]) implement an optimization for the common case of cyclic and indefinite pointers to shared memory. Cyclic pointers are the ones with a block factor of one, and indefinite pointers with a block factor of zero. Therefore, their phases are always zero. These shared pointers are thus phaseless, and the compiler exploits this knowledge to schedule more efficient operations for them. All internal shared arrays of the UPCBLAS routines are declared with block factor of zero in order to take advantage of this optimization.

- Efficient array-based reduction in UPC: As explained in Section 2.1.2, the column distribution for *Tgemv* needs a reduction operation for each element in the result vector (see Figure 2.5). The UPC collectives library [112] does not include a collective function to perform reduction operations on arrays as, for instance, MPI does. A solution could be the use of the `upc_all_reduce` function once per element of the destination array, but this method is quite inefficient. The approach followed in UPCBLAS to perform these array-based reductions consists of copying all the elements to a thread, using this thread to perform the operation and distributing the results again among the threads. We have proven experimentally that this approach is faster. This ad hoc array-based reduction was developed with an interface so that it can be easily changed if in the future a better version is included in the collectives library (e.g. a tree level reduction).

- Efficient broadcast communication model: As in the PGAS programming model any thread may directly read or write data located on a remote processor, two possible communication models can be applied to the broadcast operations:

  - Pull Model: The thread that obtains the data to be broadcast writes them in its shared memory. The other threads are expected to read them from this position. This approach leads to remote accesses from different threads but, depending on the network, they can be performed in parallel.
  - Push Model: The thread that obtains the data to be broadcast writes them directly in the shared space of the other threads. In this case the network contention decreases but the writes are sequentially performed.

  The pull communication model was initially proved to be more efficient than the push one, particularly as the number of threads increases. This is therefore the communication model used by all the broadcast operations in our parallel routines (see, for instance, the BLAS2 triangular solver in Figure 2.8 and Algorithm 2.1).

- Efficient underlying sequential numerical libraries: The UPCBLAS parallel functions call internally BLAS routines to perform the computations with local data in each thread. These calls can be linked to very optimized libraries such as Intel MKL. UPCBLAS internally includes wrappers that allow linking to libraries such as Cray LibSci or AMD ACML that do not follow the standard C BLAS interface [11] and are only able to work with column-ordered matrices.

Apart from these general techniques, other optimizations that take into account some hardware parameters are implemented for some routines. These additional optimization techniques are explained in the next chapter after presenting Servet, the benchmark suite used to know the hardware characteristics of the target system.

Chapter 3

Servet: Measuring Hardware Parameters

The growing complexity in computer system hierarchies due to the increase in the number of cores per processor, levels of cache and number of processors per node, as well as the high-speed interconnects, demands the use of new optimization techniques and libraries that take advantage of their features.

Servet[1], a suite of benchmarks focused on detecting a set of hardware parameters with high influence on the overall performance of multicore systems, is presented in this chapter. These parameters will be used to increase the performance of the UPCBLAS routines on multicore clusters. This chapter begins in Section 3.1 with an introduction and a description of the state of the art related to the automatic detection of hardware parameters. Subsequent sections describe the benchmarks included in Servet: Section 3.2 explains the cache size estimator; Section 3.3 presents the process to detect shared caches; the benchmark to find possible memory access overheads is described in Section 3.4, and Section 3.5 shows the mechanism to detect the different communication layers. Section 3.6 illustrates the algorithms integrated in Servet that provide mapping policies that automatically take into account the hardware parameters detected by the suite. Finally, Section 3.7 describes

[1]As this suite dissects the machines to discover their characteristics, it obtains its name from Miguel Servet, a Spanish theologian, physician, cartographer and humanist who lived in the XVIth Century and performed many dissections, being the first European to describe the function of pulmonary circulation.

the interaction between Servet and UPCBLAS and how the benchmark suite helps to optimize the numerical routines.

3.1. Introduction to the Detection of Hardware Parameters

The popularity of autotuned codes, which adapt their behavior to the machine where they are executed, has increased over the past few years thanks to the performance improvement that they are able to achieve. For instance, a widespread autotuning technique for sequential codes consists in using a wide search mechanism to find the most suitable algorithm [42, 88, 120]. The search time could be reduced by knowing the values of some parameters of the system [41, 124].

As regards parallel codes, there exist in the literature many optimization techniques to improve their performance, most of them directed towards increasing the communication bandwidth among the processes [98, 108, 127]. Among the different parallel architectures, clusters of multicores are nowadays the target architecture for autotuned codes. On the one hand, they usually present several non-uniform latencies and bandwidths depending on the cores that are communicating [77]. On the other hand, the overall memory access throughput might decrease if several cores share memory or cache.

Two main approaches are followed to improve the performance of parallel applications on clusters of multicores. The most common one consists in implementing and timing several codes in order to choose the best one according to the system characteristics, for instance, adapting the communication algorithms to the target machine. In [115] Vadhiyar et al. present a thorough evaluation of the MPI collectives that proves that the optimal algorithm and the optimal buffer size for a given message size depends on the gap values of the networks, the memory models and the underlying communication layer. The optimal parameters for a particular system are experimentally determined. Cuenca et al. [27, 28] have developed a mechanism to make a good automatic choice of configurable parameters for linear algebra routines. Faraj et al. [38, 39] present an automatic generation and tuning system for MPI collective communication routines and offer a successful evaluation of its impact on the NAS Parallel Benchmarks (NPB) [78]. Their approach focuses on optimizing collectives taking into account the network topology.

The second approach consists in assigning processes or threads to specific cores to improve performance without source code modifications. In [19] Chen et al. propose a profile-guided approach to optimize parallel process placement in SMP clusters, experimentally proving that it can obtain good speedups. A more complex approach that designs mapping policies which minimize network contention on large supercomputers is presented in [57]. Mercier and Clet-Ortega [74] show another evaluation (restricted to MPI) for several mapping policies using the NPB benchmarks. This work also includes a study about the influence of the shared caches on the mapping policies. In [13] Broquedis et al. propose a hierarchical approach to the execution of OpenMP threads on multicore machines, providing multicore-aware and memory-aware scheduling policies. It relies on the tool hwloc [14], which can only obtain the characteristics of the machine when available from the system specifications.

In this scenario some knowledge of the memory system hierarchy, as well as some hardware parameters, are required for every optimization effort. However, system parameters and specifications are usually vendor-dependent and often inaccessible to user applications. For instance, the administration tool *dmidecode* reports information about the hardware as described in the BIOS, but it is restricted to system administrators. Therefore, estimation by benchmarks is the only general and portable way to find out the hardware characteristics, without worrying about the vendor, the OS or the user privileges. Besides, this approach provides experimental results about the performance of the systems, obtaining a more reliable estimate than inferring them from the machine specifications.

The automatic extraction of system parameters to support the autotuning of parallel applications is a topic which has been previously tackled with different approaches. The X-Ray tool [126] provides micro-benchmarks to automatically obtain some characteristics of the CPU and the cache hierarchy for unicore processors. In [125] Yotov et al. present a new methodology to obtain the cache parameters. However, they reported some issues measuring the characteristics of the cache levels lower than L1 because they are physically indexed. In their benchmarks contiguous memory allocation is compulsory and therefore they must request virtual memory backed by a superpage. As the way to request it depends on the OS, their approach is not portable. P-Ray [35] extends X-Ray benchmarks to detect the characteristics of multicore systems. Although it is very appropriate for SMPs, it inherits most of the issues of X-Ray (e.g., non-portable L2/L3 cache parameters determination) and lacks some relevant parameters for multicore clusters like communication overhead. Although the algorithm to detect the processor mapping in P-Ray could be used (with manual work) to show different communication latencies on multicore systems, it is restricted to shared memory systems, so it is not applicable to clusters.

Servet improves previous works by providing a portable estimate of several parameters of the machine architecture such as the cache size (L1, L2 and, when available, L3 caches), cache hierarchy (determining if a cache is private to a particular core or shared among several cores), shared memory access bandwidth, shared memory hierarchy (cores that share memory and NUMA system hierarchy), and communication overheads. Servet also provides an API with C functions so that other applications can easily access the hardware parameters identified by this tool (see Appendix B).

3.2. Cache Size Estimate

The knowledge of the cache size is used in many optimization techniques to tune sequential and parallel codes, in order to minimize the number of cache misses and therefore increase the memory access performance.

Several benchmarks that estimate the cache size have been proposed during the last years. We have followed the approach of Saavedra and Smith [90]: measuring the number of cycles used to traverse arrays of different sizes using 1KB strides. This stride has been chosen because it is big enough to avoid influences of the hardware prefetcher on the measured number of cycles, as current prefetchers work with strides up to 256 or 512 bytes. It is also larger than any existing cache line size and it is a divisor of any cache size.

However, this approach presents several drawbacks [125]: the results may be disturbed by unintended optimizations of aggressive compilers and they must be interpreted to determine the different cache sizes. Our algorithm improves this approach using values read from an array as stride, thus avoiding aggressive compiler optimizations and providing directly the cache sizes.

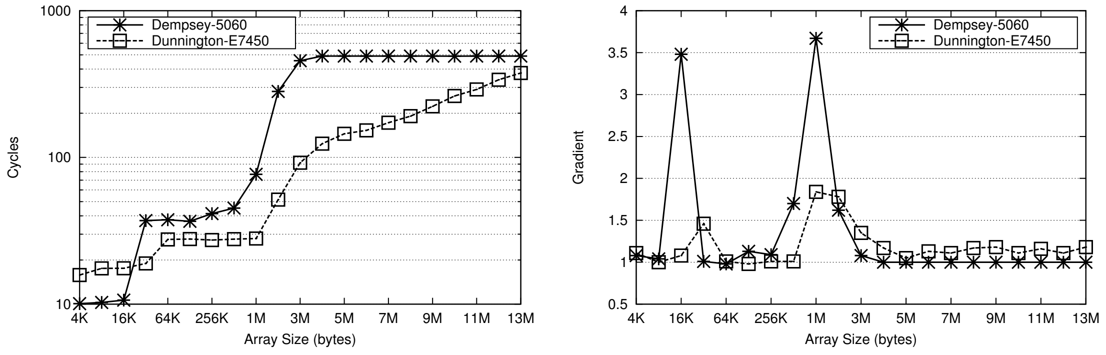
The benchmark used (*mcalibrator*) is shown in Algorithm 3.1. The outputs are two arrays `S` and `C`, of length `n`, containing the sizes of the traversed arrays and the average number of cycles required by each access during their traversal, respectively.

Algorithm 3.1 *mcalibrator* algorithm

```
aux = 0  // Auxiliary variable to help to avoid compiler influences
i = MIN_CACHE
n = 0 // Number of cache sizes tested
while i≤MAX_CACHE do
    S[n] = i  // Size of the traversed array
    size = The number of integers stored in S[n] bytes
    for j=0;j<size;j=j+1 do
        A[j] = The number of integers stored in 1KB  // Each position
        keeps the stride
    end
    // The access to the array is in the loop to know the stride
    for j=0;j<size;j=j+A[j] do
        aux = aux + size  // A variable update to avoid compiler
        optimizations
    end
    C[n] = The number of cycles to perform the previous loop
    n=n+1
    if i<2MB then
        i=i*2
    else
        i=i+1MB
    end
end
```

Figure 3.1(a) presents the cycles obtained with this algorithm on two Intel Xeon based architectures (*Dempsey*-5060 and *Dunnington*-E7450), whereas Figure 3.1(b) shows the gradient of the previous results, that is $C[k+1]/C[k], 0 \leq k < n$. These architectures will be used to explain the algorithm to detect the cache hierarchy from the number of cycles to access memory and their corresponding gradients.

Traversing an array with all its elements in cache is faster than traversing one which does not fit. Therefore, the sizes where the number of cycles rises indicate that they do not fit in cache and cache misses appear. Sharp changes in the slope



(a) Cycles needed to traverse an array (b) Gradient of the rise of cycles

Figure 3.1: *mcalibrator* results

of the cycles graph correspond to peaks in the gradients one. The L1 cache size is determined by the first peak of the gradients: 16KB for *Dempsey* and 32KB for *Dunnington* according to Figure 3.1(b).

Unfortunately, the approach followed to estimate the L1 cache size cannot always be applied to lower levels. L1 caches are typically virtually indexed, but lower levels are always physically indexed. This is a problem when the cache size is larger than one page because contiguity in virtual memory does not imply adjacency in physical memory, which leads to the generation of misses in tests with arrays much smaller than the cache considered. Therefore, peaks in the gradient function might not be enough to estimate the cache size, as for *Dempsey*, with high gradient values in the range [512KB,2MB]. Although some OS solve this problem applying page coloring, others such as Linux, widely used in scientific computing, do not.

Our benchmark overcomes this problem following a probabilistic approach. Since the OS can map a virtual page to any physical page, no assumptions can be made on which cache sets of a physically indexed cache correspond to a given virtual page. Now, in a K-way physically indexed cache of size CS with a page size PS every cache way can be divided in $CS/(K * PS)$ page sets, that is, groups of cache sets that can receive data from the same page. As a result, if the probability that a given virtual page is mapped to a given page set is uniform, the number of pages X per page set belongs to a binomial $B(NP, (K * PS)/CS)$, where NP is the number of pages involved in the access [41]. Since each set can contain up to K pages without conflicts, the probability $P(X > K)$ is the expected miss rate during the repeated access to the NP pages.

Thus, based on the outputs of *mcalibrator* (Algorithm 3.1), Algorithm 3.2 can be used to determine L2 and L3 (if present) cache size according to the previous reasoning. The probabilistic algorithm starts calculating the number of pages and the miss rate for each *mcalibrator* result. After that, it calculates the divergences between the measured and the predicted miss rates according to the binomial distribution. The estimated cache size is the one with the highest similarity (less divergence).

Algorithm 3.2 Probabilistic algorithm to determine the size of physically indexed caches (L2, L3) based on *mcalibrator* outputs

```
$hit\_time = MIN(C)$;
$miss\_overhead = MAX(C) - MIN(C)$
for $i=0;i<n;i=i+1$ do
    $MR[i] = (C[i] - hit\_time)/miss\_overhead$ // Miss Rate
    $NP[i] = S[i]/PS$ // Number of Pages
end
foreach tentative cache size CS and associativity K do
    $div[CS][K] = 0$
    for $i=0;i<n;i=i+1$ do
        $div[CS][K] = div[CS][K] + \mid MR[i] - P(X > K) \mid,$
                    $X \in B(NP[i], (K * PS)/CS)$
    end
end
Result: The statistical mode of CS using the five elements of div with the
        lowest values
```

The accuracy of this algorithm is higher than that of only searching peaks in the gradients of the *mcalibrator* outputs. For instance, in the *Dempsey* case, a 1MB L2 cache would be erroneously estimated looking at the *mcalibrator* outputs. The estimate of the latter algorithm analyzing the [256KB,4MB] range is 2MB, the correct value.

Besides, this algorithm is able to provide a correct cache size estimate when gradients are higher than 1 for a wide size range. This is the case of *Dunnington* (see Figure 3.1): the use of the algorithm in the range [3MB,13MB] provides 12MB as result, which is the actual L3 cache size.

The overall way to detect the number of cache levels and their sizes is presented

Algorithm 3.3 Algorithm to detect the cache levels and their sizes

```
Data: mcalibrator outputs from MIN_CACHE to MAX_CACHE
foreach peak in the gradients do
    if this is the first peak then
        Estimate the L1 cache size using the peak value
    else
        if peak is related only to a single array size then
            Estimate the corresponding cache size using the peak value
        else
            Estimate the corresponding cache size from the probabilistic
            algorithm using mcalibrator outputs where gradient is larger than
            1 around the peak
        end
    end
end
if the largest array sizes show a gradient > 1 then
    The corresponding cache size is the estimate of the probabilistic
    algorithm using mcalibrator outputs with the largest sizes
end
```

in Algorithm 3.3. As L1 caches are virtually indexed, their size is always calculated using the first peak of the gradients. However, there are two ways to estimate the size of the next levels. A peak clearly located only in one array size means that the OS has used page coloring so the behavior of the cache is analogous to that of a virtually indexed one and the position of the peak determines the cache size. However, a peak with high gradients for several array sizes needs the use of the probabilistic algorithm. Therefore, this benchmark is completely portable, being independent from the application of page coloring by the OS. Finally, the probabilistic algorithm is used again if gradients are higher than 1 for the largest arrays.

3.3. Determination of Shared Caches

The knowledge of which cores share a concrete cache level can be useful in order to speed up memory accesses. On the one hand, if two processes work with the same block of data which fits in cache, mapping them to cores that share cache would improve performance, as they could exchange data using the cache. On the other hand, if they do not work with the same data, their working sets could not fit in a shared cache, leading to more replacements and misses. In this case scheduling techniques for autotuning would map the processes to cores that do not share cache in order to minimize misses.

Algorithm 3.4 shows the Servet benchmark to detect shared caches. The inputs are the number of cache levels, l, and an array of length l, CS, with the cache size per level. For each cache level i, the first step is to call *mcalibrator* using an array of size a little larger than $CS[i]/2$ and keep the result as reference. Then *mcalibrator* is invoked simultaneously on two arrays of this size in two threads, varying the cores where the threads are mapped. The chosen array size provokes that two arrays do not fit simultaneously in cache so, when cores share cache, the array created by each of them is replacing the other one and the number of cycles increases. The output is an array of lists P_{sc} (one list per cache level) with the pairs of cores with a number of cycles at least twice greater than the reference value (metric *ratio* > 2 in Algorithm 3.4) and therefore sharing cache.

Algorithm 3.4 Algorithm to determine the shared caches

```
Data: l, number of cache levels; CS[0..l-1], cache size per level
for i=0;i<l;i=i+1 do
    P_sc[i] = Empty list
    ref = Cycles obtained from mcalibrator run on one core using an array
          of size $(2/3) * CS[i]$
    foreach pair of cores in the system do
        c = Cycles obtained from mcalibrator run in parallel on the cores of
              the pair using an array of size $(2/3) * CS[i]$ in each core
        ratio = c/ref
        if ratio>2 then
            Add the pair to P_sc[i]
        end
    end
end
Result: P_sc[0..l-1]
```

3.4. Memory Access Overhead Characterization

When several cores share the main memory, performance bottlenecks may arise with concurrent memory accesses. The knowledge of these bottlenecks would allow to implement scheduling policies in autotuned applications to avoid them and improve memory access performance.

A benchmark that provides performance results of concurrent memory accesses has been developed. This benchmark is similar to the previous one, as it compares the bandwidth to memory using an isolated core (named reference value) with the one obtained when accessing by pairs of cores. This approach to calculate the bandwidth is based on similar tools that measure it, such as *STREAM* [100]. In Servet, it is the bandwidth from the copy of all the elements stored in one array to another (these arrays must not fit in cache).

Algorithm 3.5 Algorithm to characterize memory access overhead

```
n = 0  // Number of different overhead levels found
ref = Memory bandwidth when accessing with an isolated core
foreach pair of cores in the system do
    b = Memory bandwidth for one core when accessing both cores
        concurrently
    if b<ref then
        if b is similar to a given BW[i], 0≤i<n then
            Add the pair to P_m[i]
        else
            BW[n] = b
            P_m[n] = Empty list
            Add the pair to P_m[n]
            n=n+1
        end
    end
end
Result: n, BW[0..n-1], P_m[0..n-1]
```

The benchmark is shown in Algorithm 3.5. For each pair of cores, the bandwidth of one core when both of them are concurrently accessing memory is calculated and compared to the reference value i.e., the memory bandwidth when accessing with an isolated core. A bandwidth for concurrent accesses significantly lower than the reference value indicates an overhead.

However, distinguishing the different magnitudes of overhead and which pairs suffer each of them is also interesting. To do it, the algorithm works with two arrays: BW, with the different bandwidths lower than the reference value, and P_m, which contains the pairs of cores which cause each overhead ($P_m[i]$ is the list of core pairs which obtained a concurrent bandwidth similar to $BW[i]$). When a bandwidth lower than the reference value is found, the algorithm searches in the BW array if any previous pair already obtained a similar overhead. In this case, the pair which is being studied is added to the list of P_m corresponding to that bandwidth. Otherwise, this is the first pair with that specific overhead, so BW and P_m are updated appending to them a new entry with the new bandwidth and a list with the studied pair, respectively.

The groups of cores that collide accessing memory with a given overhead are easily obtained from P_m. For instance, if the list in $P_m[i]$ has the pairs (0,1),(0,2),(3,4) and (3,5), it allows identifying two groups for the overhead $BW[i]$: {0,1,2} and {3,4,5}.

This knowledge about memory access overhead of groups of cores can be used to analyze the scalability of the memory access performance. This parameter has a special importance, as codes could be optimized by limiting the number of cores accessing memory concurrently if a poorly scalable memory system is detected. Characterizing the effective bandwidth according to the number of threads that are being executed only requires one group per overhead. For instance, for the previous example, the concurrent memory access bandwidth of cores 0, 1 and 2 is the same as the one for cores 3, 4 and 5. Therefore, using the arrays BW and P_m, the effective bandwidth to memory can be characterized without using all cores.

3.5. Determination of Communication Costs

The characterization of communication costs is divided in three parts. First, communication layers (sets of pairs of cores whose communication costs are similar) are established; then, these layers are used to characterize communication performance and, finally, to evaluate the scalability of the communication system.

In order to group the cores according to their communication costs, the benchmark shown in Algorithm 3.6 has been implemented. The reference implementation uses the MPI library. It compares the latencies when sending a message between different pairs of cores. Several representative message sizes can be selected for this task. In this case, the message size is equal to the L1 cache size, because it allows finding differences in communications when sharing other cache levels.

Algorithm 3.6 Algorithm to categorize communication costs

```
n = 0  // Number of different layers
foreach pair of cores in the system do
    l = Latency sending a message between the two cores
    if l is similar to a given L[i], 0≤i<l then
        Add the pair to P_l[i]
    else
        L[n] = l
        P_l[n] = Empty list
        Add the pair to P_l[n]
        n=n+1
    end
end
Result: n, L[0..n-1], P_l[0..n-1]
```

The algorithm is similar to the previous one: for each pair of cores, it obtains the latency to send a message between them and stores the different latencies in array L. Besides, the array P_l is created so that $P_l[i]$ keeps the list of pairs with latency $L[i]$. Finally, the cores that present the same communication performance are grouped.

Once the layers are established, the followed approach is to store the performance results of a point-to-point communication micro-benchmarking (also implemented with MPI) for representative message sizes and for each representative pair of cores (one pair per layer). The communication performance for the rest of pairs is the same as for the representative pair of their group.

Finally, in order to characterize the scalability of all layers, the performance of all the cores in a given layer concurrently sending one message is compared to the latency of an isolated message. In fully scalable communication systems, times should be similar because each core only sends one message. However, many cluster interconnection networks can present performance penalties when concurrent messages are sent. The cost of sending concurrently `N` messages of size `S` is usually higher than sending one message of size `N*S`. Thus, it is possible to optimize communication performance by gathering messages in poorly scalable systems.

3.6. Mapping of Parallel Applications on Multicore Architectures Using Servet

Nowadays, most of parallel numerical codes run on clusters of multicores. Depending on the number of threads or processes required by the user and the total number of nodes and cores available in the system, there are usually many different ways to assign threads to cores. Most previous works [19, 74] try to improve performance by using process mappings that minimize the access to the interconnection networks while increasing the use of shared memory. This is also the usual approach followed by default by many OS, which assign UPC threads to cores trying to minimize the number of nodes and NUMA regions. However, many architectures present memory issues such as shared memory buses or shared caches that lead to bottlenecks when several cores in the same NUMA region or node access memory at the same time. For instance, these issues are very significant in UPC BLAS1 and BLAS2 routines which are continuously accessing memory with many more local than remote accesses.

The information about the overheads obtained by Servet can be used to map processes or threads to certain cores in order to avoid either communication or memory access bottlenecks. Even if not all the overheads can be avoided, process mappings that minimize their impact can be applied. The potential performance benefits of the use of the information provided by Servet for mapping issues have motivated the development of Algorithm 3.7, that automatically provides the placement of processes or threads to specific cores. The mapping is chosen based on the information about shared caches, overheads in the access to memory and communication layers provided by Servet. Each core in the system is assigned a weight that represents the overhead cost of its selection. Initially, all the weights are 0, which means that any core can be selected. From then on, the core with the lowest weight is chosen.

Algorithm 3.7 Algorithm to provide the best process mapping

```
foreach core c in the system do
    Weight[c]=0
end
foreach process p to map do
    Assign p to the core c with the lowest Weight[c]
    // Update the weights of the cores
    foreach cache in the system do
        foreach core c2 that shares that cache with c do
            Increase Weight[c2]
        end
    end
    foreach memory access overhead in the system do
        foreach core c2 that shares the overhead with c do
            Increase Weight[c2]
        end
    end
    foreach core c2 still not assigned do
        if latency(c,c2) < MaxLatency then
            Decrease Weight[c2]
        end
    end
end
```

Whenever a core is selected, the weights are updated according to the following rules:

1 The weights of the cores that share cache with the selected one are increased. This rule is applied for each cache level to avoid the loss of performance when shared caches lead to an increase of the number of cache misses.

2 The weights of the cores that show additional overhead when accessing memory concurrently with the selected core are increased.

3 The weights of the cores whose communication latencies with the selected one are significantly lower than the maximum latency in the system (*MaxLatency* parameter, obtained by Servet) are decreased to promote their selection. They are decreased in a magnitude that depends on the difference between the current latency and *MaxLatency*. Note that shared memory transfer optimiza-

tions in the communication libraries can be taken into account through the application of this rule.

The increase or decrease of weights applied to each one of the previous rules depends on the characterization of the code as either memory-bound or communication-intensive. Currently this characterization is provided by the user through the `SERVET_MEM_PRIOR` or `SERVET_COMM_PRIOR` parameters, respectively. In a memory-bound code the increase due to rules 1 and 2 is set as ten times larger than the decrease applied by rule 3. In communication-intensive codes the opposite practice is applied.

The operation of this mapping procedure is illustrated through 2 nodes of an x86_64 multicore cluster with InfiniBand network (20Gbps). Figure 3.2 presents the architecture of this system. Each node has 2 Intel Xeon Nehalem quadcore E5520 CPUs at 2.27 GHz and 8GB of memory. Servet has detected on this system the correct cache sizes (L1: 32KB, L2: 256KB, L3: 8MB) and topology, where the L3 cache is the only one shared, by pairs of cores. Moreover, the concurrent access to memory by two cores within the same processor presents an important overhead. Finally, the latency of inter-node communications is significantly higher than the intra-node ones.

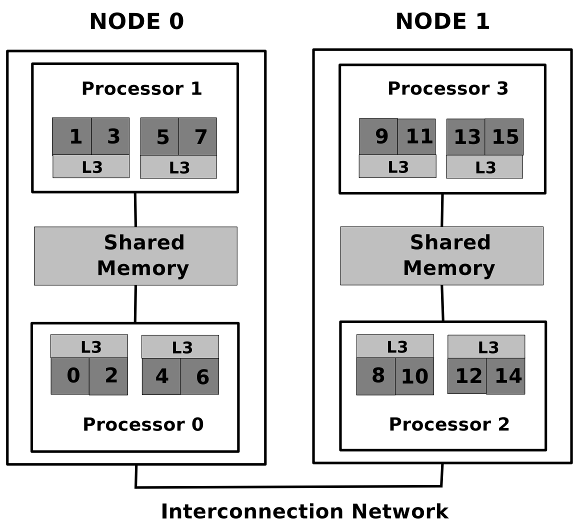


Figure 3.2: Architecture of 2 nodes of the x86_64 cluster

Tables 3.1 and 3.2 present step by step the operation of the selection proce-

Table 3.1: Evolution of the weight values for all cores in 2 nodes of the x86_64 cluster when mapping 4 processes with SERVET_MEM_PRIOR

Iter	Core →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Total	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Rule 1	-	0	*10*	0	0	0	0	0	0	0	0	0	0	0	0	0
	Rule 2	-	0	*10*	0	*10*	0	*10*	0	0	0	0	0	0	0	0	0
	Rule 3	-	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	0	0	0	0	0	0	0	0
	Total	-	***-1***	*19*	*-1*	*9*	*-1*	*9*	*-1*	0	0	0	0	0	0	0	0
3	Rule 1	-	-	0	*10*	0	0	0	0	0	0	0	0	0	0	0	0
	Rule 2	-	-	0	*10*	0	*10*	0	*10*	0	0	0	0	0	0	0	0
	Rule 3	-	-	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	0	0	0	0	0	0	0	0
	Total	-	-	*18*	*18*	*8*	*8*	*8*	*8*	**0**	0	0	0	0	0	0	0
4	Rule 1	-	-	0	0	0	0	0	0	-	0	10	0	0	0	0	0
	Rule 2	-	-	0	0	0	0	0	0	-	0	*10*	0	*10*	0	*10*	0
	Rule 3	-	-	0	0	0	0	0	0	-	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*
	Total	-	-	*18*	*18*	*8*	*8*	*8*	*8*	-	***-1***	*19*	*-1*	*9*	*-1*	*9*	*-1*

Table 3.2: Evolution of the weight values for all cores in 2 nodes of the x86_64 cluster when mapping 4 processes with SERVET_COMM_PRIOR

Iter	Core →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Total	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Rule 1	-	0	*1*	0	0	0	0	0	0	0	0	0	0	0	0	0
	Rule 2	-	0	*1*	0	*1*	0	*1*	0	0	0	0	0	0	0	0	0
	Rule 3	-	*-10*	*-10*	*-10*	*-10*	*-10*	*-10*	*-10*	0	0	0	0	0	0	0	0
	Total	-	***-10***	*-8*	*-10*	*-9*	*-10*	*-9*	*-10*	0	0	0	0	0	0	0	0
3	Rule 1	-	-	0	*1*	0	0	0	0	0	0	0	0	0	0	0	0
	Rule 2	-	-	0	*1*	0	*1*	0	*1*	0	0	0	0	0	0	0	0
	Rule 3	-	-	*-10*	*-10*	*-10*	*-10*	*-10*	*-10*	0	0	0	0	0	0	0	0
	Total	-	-	*-18*	*-18*	***-19***	*-19*	*-19*	*-19*	0	0	0	0	0	0	0	0
4	Rule 1	-	-	0	0	-	0	*1*	0	0	0	0	0	0	0	0	0
	Rule 2	-	-	*1*	0	-	0	*1*	0	0	0	0	0	0	0	0	0
	Rule 3	-	-	*-10*	*-10*	-	*-10*	*-10*	*-10*	0	0	0	0	0	0	0	0
	Total	-	-	*-27*	*-28*	-	***-29***	*-27*	*-29*	0	0	0	0	0	0	0	0

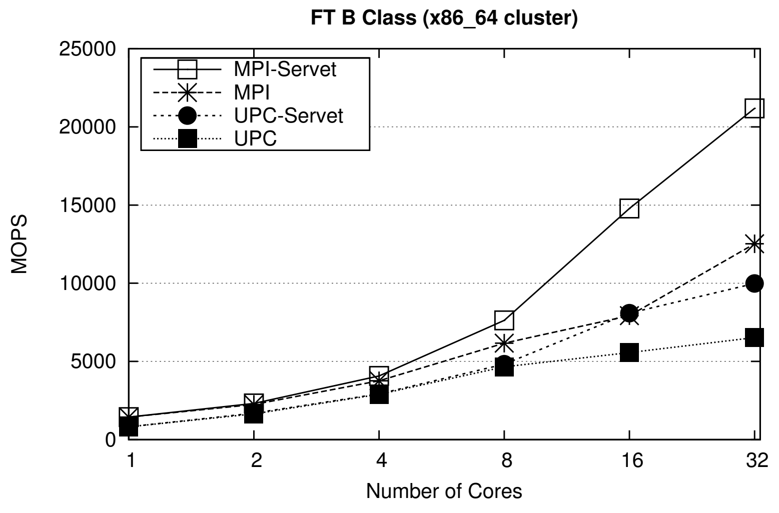
dure (following Algorithm 3.7) for mapping 4 processes to this system with the SERVET_MEM_PRIOR and the SERVET_COMM_PRIOR policies, respectively. The core selected in each iteration is in bold type in the tables. In Table 3.1, initially core 0 is selected. After mapping the first process, the second iteration starts applying the first rule, which increases the value of the core 2, as it shares L3 cache with core 0. L1 and L2 caches are not considered because they are not shared. Then, rule 2 increases the weights of cores 2, 4 and 6, because they present an overhead when accessing memory concurrently with core 0. Finally, using the third rule, the weights of the cores in the same node, whose communications are faster, are decreased. The increase due to rules 1 and 2 is ten times higher than the decrease considered for the third rule because SERVET_MEM_PRIOR was selected. Once all the rules have been applied, core 1 is selected because it is the first core with the lowest weight value. With this selection the messages between both cores (0 and 1) present low latency (according to Servet, there is no significant difference in the intra-node latencies) and all memory access overheads are avoided. As can be seen in Table 3.1, the algorithm leads to map the processes to cores 0, 1, 8 and 9. Although there would be some messages in the interconnection network because both nodes are used, this assignment reduces memory access overhead.

Table 3.2 shows the evolution of the weights for the same example but with the `SERVET_COMM_PRIOR` policy. Here the algorithm advises to place the processes in cores within the same node to avoid communications through the network (cores 0, 1, 4 and 5). In addition, inside the node, Servet chooses cores that do not share the L3 cache. However, it cannot avoid memory overheads caused by concurrent memory accesses from cores in the same processor.

This mapping algorithm was proven to be very effective for most of the NAS Parallel Benchmarks (NPB) [78]. Figure 3.3 shows two examples of the performance improvement obtained by applying the `SERVET_MEM_PRIOR` mapping policy to the UPC and MPI versions of the FT and IS benchmarks using up to 32 threads/processes on a system with 8 nodes as the ones shown in Figure 3.2. The performance metric selected is MOPS (Million of Operations Per Second) and the performance results that use the mapping policy are labeled as *Servet*. A more detailed study can be found in [53].

3.7. Use of Servet to Optimize UPCBLAS Routines

If Servet is previously installed and executed in the system, UPCBLAS can use the API described in Appendix B to obtain some hardware parameters and automatically adapt the behavior of some routines to the characteristics of the machine on which the library is installed. Servet saves the relevant hardware parameters into a text file and the API provides functions to access the information from this file. Then, when a UPCBLAS routine requires information about the hardware characteristics it resorts to calling the API of Servet instead of running any benchmark.



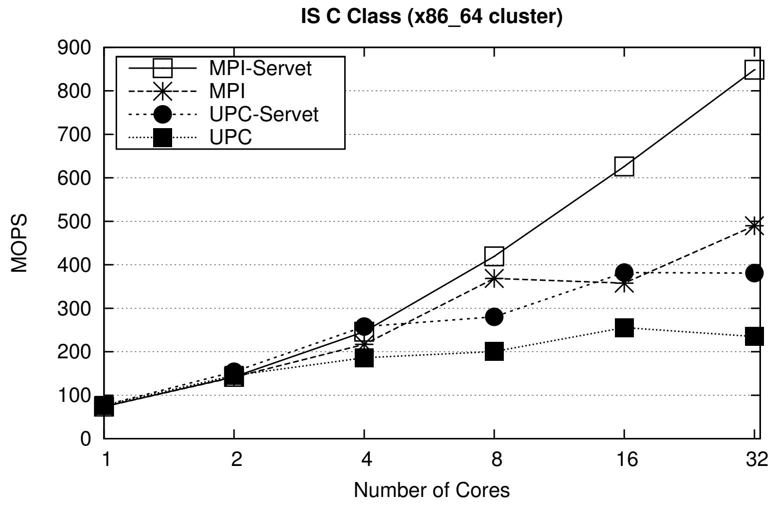


Figure 3.3: Impact of the Servet mapping policy on NPB performance on the x86_64 cluster

Thus, the performance overhead caused by Servet is almost negligible in UPCBLAS.

3.7.1. Efficient Mapping of UPC Threads

As previously explained, the correct placement of UPC threads on the cores of a cluster is key to obtaining good performance. UPCBLAS users can take advantage of the mappings provided by Servet and explained in Section 3.6 to improve the efficiency of the parallel numerical routines. As a rule of thumb, if the main part of a numerical code is performed by BLAS1 and BLAS2 routines, the mapping with the `SERVET_MEM_PRIOR` policy is the best option. The `SERVET_COMM_PRIOR` policy should be selected for BLAS3-based codes. A study of the performance improvement thanks to the Servet mappings will be shown in Chapter 4.

3.7.2. On-Demand Copies in BLAS3 Routines

In many of the UPCBLAS routines all threads have to access all the elements of a distributed matrix or vector (see, for instance, vector `x` in Figure 2.4 or matrix `B` in Figure 2.9). Thus, all threads must copy remote data to local memory before performing the numerical computations with their local data. Auxiliary buffers in local memory are required to store the vector or the matrix.

In the UPC BLAS2 routines the vector is stored completely in private memory and then all the numerical computations related to a thread are performed in one go. However, copying the whole matrices in the BLAS3 routines could involve important memory overheads because the buffer could need to allocate a huge amount of private memory. Moreover, performance would be affected because all threads should wait to copy all these data before starting the sequential computations. Besides, they would access a large amount of remote data at the same time, which could lead to network contention in many systems.

In order to overcome these drawbacks the UPC BLAS3 routines are implemented using what has been called an *on-demand copies* technique. The matrix is copied by blocks into the auxiliary buffer, decreasing memory requirements. Once one block is copied, the internal numerical computation that uses that part of the matrix can start. As well, computations and communications are overlapped by using split-phase barriers and the asynchronous copies presented at the end of Section 1.2.

The size of the internal blocks is very important in this approach. On the one hand, if the blocks are too large, the memory and performance problems explained before would not be solved. On the other hand, if they are too small the performance of the communications would decrease because more calls to `upc_memget()` would be needed, each of them with less aggregation of remote accesses; moreover, the partial sequential computations would be almost negligible to be overlapped with communications.

The best size can be different depending on the characteristics of the system where the library is executed, specially on the communication network. UPCBLAS uses the information about communication layers and their bandwidths provided by Servet to determine the message size for which the bandwidth stops increasing. This message size is considered as the ideal block size to use in the on-demand copies technique. For instance, Figure 3.4 shows the bandwidths of the intra-node and inter-node communications on Carver, an IBM supercomputer with Intel Xeon 5550X (Nehalem) processors and InfiniBand network that will be presented in Section 4.1 for the experimental evaluation. The size selected for the auxiliary buffer in this system is 2MB because it is the point where both bandwidths no longer show an increase.

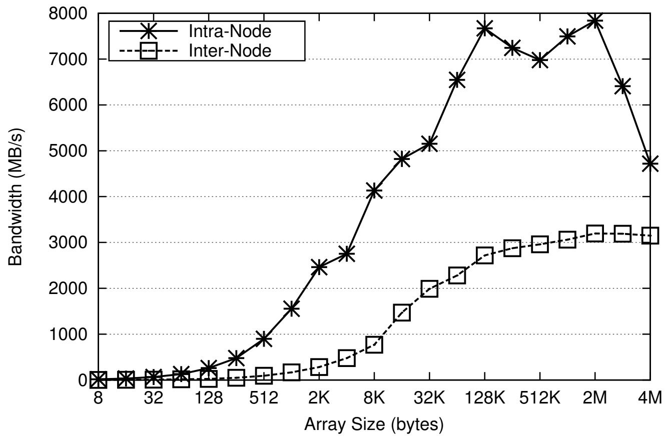


Figure 3.4: Example of the information provided by Servet about the communication bandwidth as a function of the message size on the Carver supercomputer

Chapter 4

Experimental Evaluation

This chapter provides an extensive performance evaluation of UPCBLAS using two clusters with very different hardware characteristics, underlying numerical libraries and UPC compilers. The UPCBLAS routines are benchmarked by calling them with different input parameters in order to determine which options obtain the best performance (e.g. row or column distribution, different `block_size`...). An evaluation of the impact of the optimization techniques using Servet (see Section 3.7) is also included. Furthermore, this chapter also explains the implementations of Cholesky and LU solvers which take advantage of UPCBLAS routines to perform part of the computation. These implementations prove that the usage of the library to develop numerical codes is very easy and intuitive. The performance of both the isolated UPCBLAS routines and the presented factorizations is compared to similar MPI counterparts.

The structure of this chapter is as follows: Section 4.1 introduces the two experimental testbeds where the benchmarks were executed. The benchmarking of the UPCBLAS routines, including the study of the impact of the optimization techniques, the evaluation of different algorithms and the comparison with MPI routines, is addressed in Section 4.2. Finally, Section 4.3 describes how UPCBLAS can be used to facilitate the development of Cholesky and LU factorizations, as examples of more complex numerical codes. A performance evaluation and comparison with MPI counterparts is also included.

4.1. Experimental Testbeds

Two machines from different vendors with different architectures, interconnection networks, UPC compilers and underlying libraries with BLAS routines were used in order to evaluate the performance of UPCBLAS in various scenarios:

- **Carver IBM iDataPlex** [17]. It is installed at the National Energy Research Supercomputing Center (NERSC). This system consists of 320 nodes, each of them with 2 quad-core Intel Xeon 5550X (Nehalem) processors (8 cores at 2.67 Ghz per node) and 24GB of memory. L1 and L2 caches are 16KB and 256KB independent caches, respectively, while L3 is an 8MB cache shared by the four cores in each processor. The compute nodes are interconnected by a 4X QDR InfiniBand network (32GBps of theoretical effective bandwidth). As for software, the code was compiled using Berkeley UPC 2.14.2 [10] with icc 12.1.3 as underlying C compiler. The inter-node communications are performed through GASNet over InfiniBand and UPCBLAS is linked to the Intel Math Kernel Library (MKL) version 10.2.2 [62], a library with highly tuned BLAS routines for Intel processors, to perform the numerical computations with the local data within each thread. The implementation of the PBLAS routines included in this MKL library was also used for comparison purposes.
- **HECToR Cray XE6** [55]. This is a supercomputer with 90,112 cores and 90TB of memory ranked 32^{nd} in the June 2012 TOP500 list [110]. The system consists of 2,816 nodes, each of them with two AMD Interlagos processors with 16 cores (grouped in two NUMA regions, with 8 cores each) at 2.3 GHz and 16GB of memory. The cache topology is quite complex: each core has an independent 16KB L1 cache, while the L2 (2MB) and L3 (8MB) caches are shared by groups of four and eight cores, respectively. Inter-node communications are performed through the custom Cray Gemini Network, which is a high-bandwidth and low-latency 3D torus interconnect with RDMA hardware support that facilitates communication overlapping. In this system UPCBLAS was compiled with the Cray UPC compiler available in the Cray CC compiler 8.0.4, and Cray LibSci version 10.0.06 was used as underlying numerical library. UPCBLAS routines were also compared to the PBLAS ones included in LibSci.

Table 4.1 summarizes the most important features of both systems. The access to these large supercomputers was possible thanks to two research visits to LBNL (USA) and EPCC (UK). As the number of available CPU hours was limited, representative numbers of cores (and not the maximum) were used to study the performance of the routines. Although some experiments do not use all cores per node and all nodes available in the system, the performance evaluation was carried out in a real environment with almost 100 of the remaining cores running jobs of other users. Additional experimental results on the Finis Terrae supercomputer [40], an HP cluster based on Intel Itanium2 processors (described in Section 5.5) can be found in [51].

Table 4.1: Summary of the testbeds used for benchmarking

System →	Carver	HECToR
Number of nodes	320	2,810
Cores per node	8	32
Processor	Intel Xeon	AMD Interlagos
Interconnection network	InfiniBand	Cray Gemini
UPC Compiler	Berkeley UPC	Cray UPC
Underlying numerical library	MKL	LibSci

4.2. Benchmarking of Isolated Routines

In this section performance evaluations of representative UPCBLAS routines are shown, specifically: the dot (*Tdot*), matrix-vector (*Tgemv*), matrix-matrix (*Tgemm*) and outer (*Tger*) products, as well as the BLAS2 and BLAS3 triangular solvers (*Ttrsv* and *Ttrsm*). The performance of the six routines was measured for single precision using different distributions. A comparison with the MPI implementation of the PBLAS routines [21, 83] is also provided. The PBLAS routines were tested using different data distributions (by rows, by columns and 2D distributions, using different block sizes), but the results presented in this section are those of the distribution that achieves the best execution time for each PBLAS routine and number of processes.

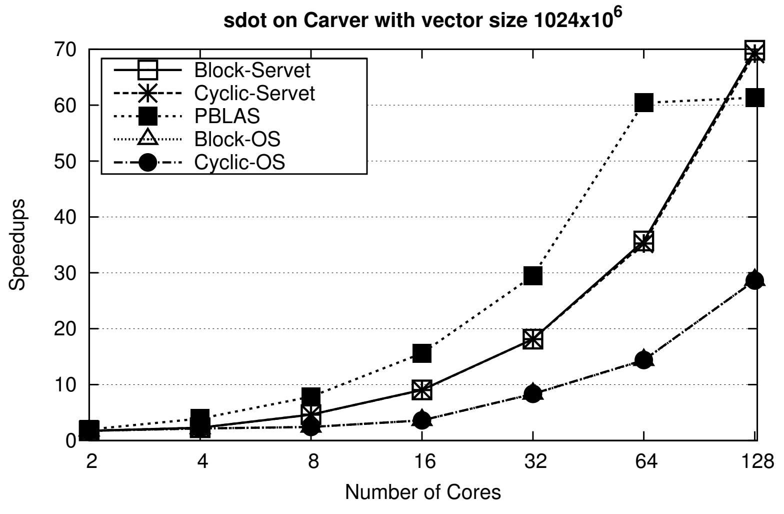
The sizes of the vectors and matrices used in the experiments are the largest ones that can be allocated in the memory available for one core (in order to calculate speedups). All the speedups (both for UPCBLAS and PBLAS routines) were obtained using as reference the execution times of the routines of the sequential library. The experiments with BLAS1 and BLAS2 routines were repeated 20 times both for UPCBLAS and PBLAS. As the evaluation was performed in a non-dedicated environment, the results shown in the following graphs are always the best ones for each library and scenario, thus discarding most sources of variability. As the execution times for the BLAS3 routines are much larger, the influence of the variability on the calculation of speedups is less significant. Therefore, in this case the experiments were performed only 3 times.

Most of the conclusions were taken using the common strong scaling speedups. In this case all the experiments were performed using matrices of the same size and the speedups reflect how much the parallel routine is faster than the sequential counterpart completing exactly the same operation. They are calculated as T_1/T_n, T_1 being the sequential execution time and T_n the parallel execution time when using n cores.

Furthermore, a weak scaling study, where the size of the input vector and matrices increases with the number of cores, is also included. As parallel algorithms with many cores are generally used to solve problems that cannot be performed with less resources (e.g. not enough memory), weak scaling allows evaluating the behavior of the algorithms in a more realistic scenario than strong scaling. In this study the average computational workload (number of floating point operations) per thread was fixed for all experiments. The weak scaling speedup is calculated as $(T_1/T_n) * n$ and thus programs with perfect scalability would obtain the same execution times for any number of cores.

4.2.1. BLAS1 Routines

Figure 4.1 and Tables 4.2 and 4.3 show the speedups and execution times for the BLAS1 dot product. Experimental results have been obtained with strong scaling and using the cyclic (i.e. `block_size` = 1) and block-cyclic distributions (with different `block_size` values) in order to analyze the behavior of the routine in several



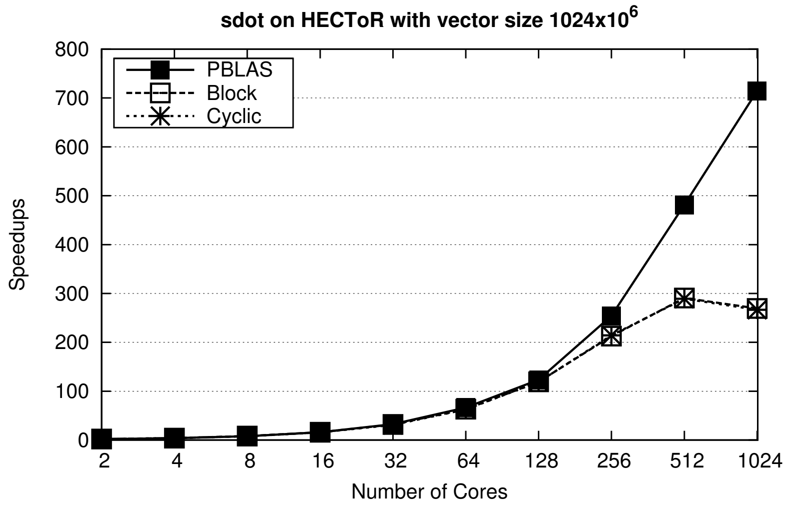


Figure 4.1: Strong scaling speedups of the single precision dot product (*sdot*)

scenarios. For illustrative purposes only the "extreme" distributions are presented throughout this section (i.e. cyclic and block). The scalability is reasonable for both distributions taking into account that the execution times are very short. An analysis of the results obtained with different values for `block_size` has demonstrated that it has no influence on the performance of this routine. Therefore, UPCBLAS users can achieve the best performance independently of the `block_size` used in the application. In general this also applies to the BLAS2 and BLAS3 products.

Furthermore, two different results for each UPCBLAS distribution on Carver are

Table 4.2: Strong scaling execution times (in milliseconds) of the single precision dot product (*sdot*) on Carver

sdot with size 1024×10^6 (ms)					
	Cyclic		Block		PBLAS
Cores ↓	OS Map	Servet Map	OS Map	Servet Map	
Seq	641.66				
2	367.93	367.93	367.66	367.66	326.55
4	296.93	281.48	297.07	281.60	163.65
8	264.96	138.87	265.18	138.83	81.67
16	178.23	71.12	178.03	70.80	41.08
32	77.01	35.39	77.05	35.40	21.78
64	44.60	18.20	44.70	17.98	10.62
128	22.42	9.27	22.39	9.19	10.46

Table 4.3: Strong scaling execution times (in milliseconds) of the single precision dot product (*sdot*) on HECToR

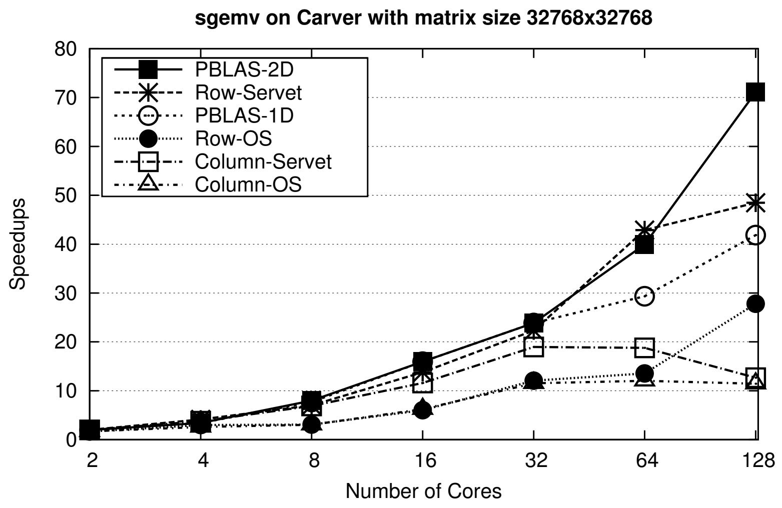
sdot with size 1024×10^6 (ms)			
Cores ↓	Cyclic	Block	PBLAS
Seq	1136.04		
2	579.32	579.02	614.05
4	287.31	287.46	284.65
8	144.20	142.25	144.34
16	73.17	71.07	71.02
32	36.61	35.56	34.90
64	18.19	17.94	17.05
128	9.57	9.53	9.21
256	5.30	5.34	4.47
512	3.93	3.91	2.36
1024	4.26	4.22	1.59

shown: one letting the OS automatically map UPC threads to cores (labeled as `OS`) and another with the `SERVET_MEM_PRIOR` mapping policy presented in Section 3.6 (labeled as `Servet`). Servet detected that there is a memory overhead when two or more cores in the same quad-core processor try to access memory at the same time. Thus, its scheduling policy tries to minimize the number of threads per processor for the BLAS1 and BLAS2 routines (with intensive memory access), significantly outperforming the OS mapping, that always tries to place threads as close as possible to one another. For instance, the Servet mapping obtains over 3 times better performance using 128 cores. It must be remarked that on Carver the optimized mapping policy provided by Servet is used in all PBLAS executions for this routine and the BLAS2 ones. On HECToR Servet does not detect any memory overhead and thus only results using the OS mapping policy are shown for each UPCBLAS distribution.

Looking at Figure 4.1, results look much better on HECToR than on Carver, as speedups are higher. However, these better speedups do not lead to shorter execution times for the same number of cores, as can be seen in Tables 4.2 and 4.3. The reason is that the sequential dot product in MKL is much faster than in LibSci (an issue that will occur with BLAS2 routines as well). Thus, the unavoidable overhead introduced by the final reduction operation has less influence on the speedup calculation on HECToR. In general, the performance of the PBLAS routine is better than UPCBLAS for most scenarios, especially on HECToR. On this system the final MPI reduction operation is better optimized for a large number of threads than the Cray UPC reduction. Nevertheless, the differences in terms of execution times between the PBLAS and the best UPCBLAS dot product are not considered significant (less than 3 milliseconds for the worst case). On Carver the PBLAS routine stops scaling at 128 cores. Thus, the UPC implementation can be assumed to outperform the MPI one for larger experiments on this machine (as happens for 128 cores).

4.2.2. BLAS2 Routines

Experimental results for the strong scaling of the matrix-vector product (*gemv*) are collected in Figure 4.2 and Tables 4.4 and 4.5. Results for two representative matrix distributions, row cyclic and column cyclic, are shown. Further experiments with block and block-cyclic distributions were also performed (by varying the `block_size` parameter), but they were not included in the tables and graphs as the variation of performance was not significant. The `sec_block_size` parameter in UPCBLAS calls that offered the best performance was selected in order to provide a fair comparison with PBLAS, where only results with the best distribution are shown. As memory accesses are the main bottleneck in BLAS2 routines, experimental results with the OS and the `SERVET_MEM_PRIOR` policies are shown for Carver.



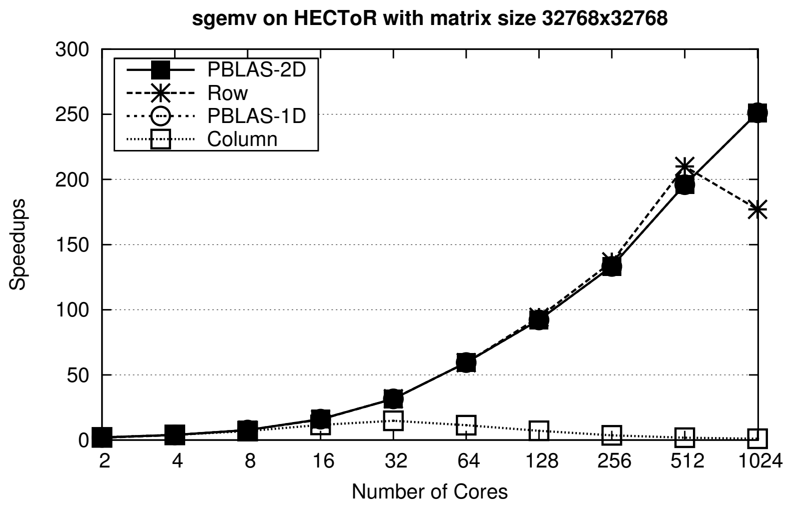


Figure 4.2: Strong scaling speedups of the single precision matrix-vector product (*sgemv*)

Comparing the two distributions used, the reduction operation at the end of the routine when using the column distribution (see Section 2.1.2 and Figure 2.5) decreases performance and even leads the function to stop scaling at 32 cores independently of the machine. In contrast, the scalability of the row distribution is high on both systems considering the short sequential times. For instance it scales up to 512 cores on HECToR even though the sequential time is only around one second.

Two experimental results were obtained for each PBLAS routine. The line la-

Table 4.4: Strong scaling execution times (in milliseconds) of the single precision matrix-vector product (*sgemv*) on Carver

sgemv with matrix size 32768x32768 (ms)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	OS Map	Servet Map	OS Map	Servet Map	1D	2D
Seq	460.052					
2	276.65	231.76	275.15	232.50	236.77	236.77
4	153.96	111.35	177.71	122.68	134.74	134.74
8	150.16	65.53	150.92	66.83	59.712	57.72
16	76.58	33.37	73.81	39.70	28.85	28.85
32	38.16	20.60	39.89	24.28	19.29	19.29
64	34.00	10.73	38.23	24.52	15.69	11.52
128	16.54	9.49	40.23	36.32	10.99	6.47

Table 4.5: Strong scaling execution times (in milliseconds) of the single precision matrix-vector product (*sgemv*) on HECToR

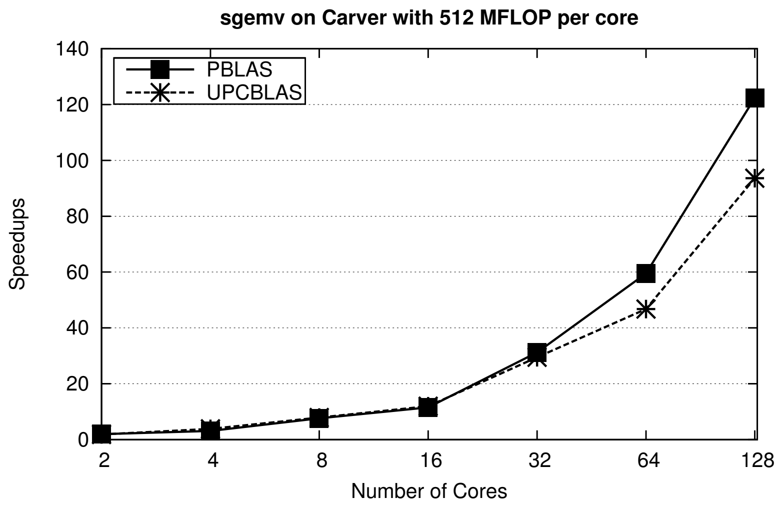
sgemv with matrix size 32768x32768 (ms)				
Cores ↓	Row Dist	Column Dist	PBLAS-1D	PBLAS-2D
Seq	678.071			
2	364.36	341.60	339.36	339.36
4	176.71	172.27	169.68	169.68
8	89.57	99.31	87.86	87.86
16	42.85	58.27	42.33	42.33
32	21.40	45.92	21.46	21.46
64	11.42	59.49	11.41	11.41
128	7.20	95.92	7.36	7.36
256	4.96	185.06	5.09	5.09
512	3.23	375.66	3.46	3.46
1024	3.83	656.56	2.70	2.70

beled as `PBLAS-2D` shows the best results for the MPI routine, which are usually obtained with 2D data distributions. However, as the UPC algorithms can only use 1D data distributions, the MPI results using the best 1D distribution, labeled as `PBLAS-1D`, are also shown for comparison purposes. It must be remarked that the 1D distribution is a particular case within the 2D distribution. Thus, scenarios where the `PBLAS-1D` and `PBLAS-2D` execution times are the same means that the best PBLAS performance is obtained with the 1D distribution. This is especially common for experiments using few cores.

The comparison with the MPI approaches proves that the UPC algorithm by rows is well implemented and the optimization techniques are effective: for example, it obtains similar of better speedups than `PBLAS-1D` up to 64 cores on Carver. Besides, it is competitive compared to the more complex `PBLAS-2D` approach. The better `PBLAS-2D` performance for 128 cores is due to the fact that, as explained in Section 2.1, this library sacrifices ease of use and productivity for performance by using complex data structures to be able to work with 2D distributions. Surprisingly, on HECToR the LibSci library has a version of PBLAS *sgemv* that is based on a 1D algorithm by rows. Thus, it does not obtain any advantage over using 2D distributions and all `PBLAS-1D` and `PBLAS-2D` execution times are the same in Table 4.5. UPCBLAS performance is, in general, similar or better than the PBLAS counterpart on HECToR. The only exception is the execution with 1024 cores where the UPCBLAS routine stops scaling due to the time needed to replicate the whole vector `x` to all threads (see Figure 2.4). The PBLAS library can perform this replication with the specific `MPI_Allgather` collective that allows PBLAS to continue scaling at 1024 cores. However, as there is no similar collective for UPC, each thread needs to perform several remote copies within the UPCBLAS product, making it less efficient than the MPI collective.

Figure 4.3 shows summarized results for weak scaling. In these graphs only the best approaches for PBLAS and UPCBLAS are represented. The main difference with strong scaling is that speedups are significantly higher (for both PBLAS and UPCBLAS libraries), demonstrating that these routines are specially suitable for large problems, where the potential of the system is better exploited. An additional remarkable feature is that the UPCBLAS routine scales up to 1024 cores on HECToR. Finally, despite UPCBLAS is always very competitive, the comparison is less beneficial than for the strong scaling experiments because the replication of vector `x` for a large number of threads is more expensive, as the length of the vector grows with the number of threads. Therefore, the positive impact of the `MPI_Allgather` collective is more significant and PBLAS always outperforms UPCBLAS. The difference between both libraries will narrow as soon as this collective will be included in the UPC collectives library.

Figure 4.4 and Tables 4.6 and 4.7 illustrate the performance of the outer prod-



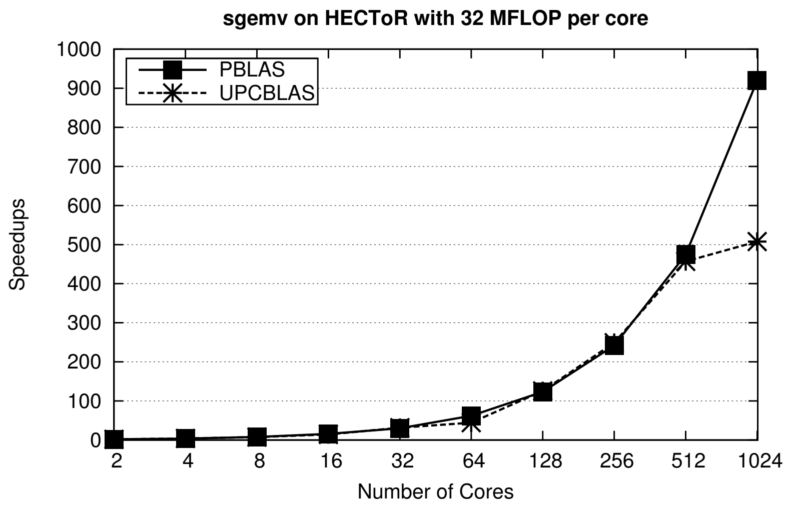
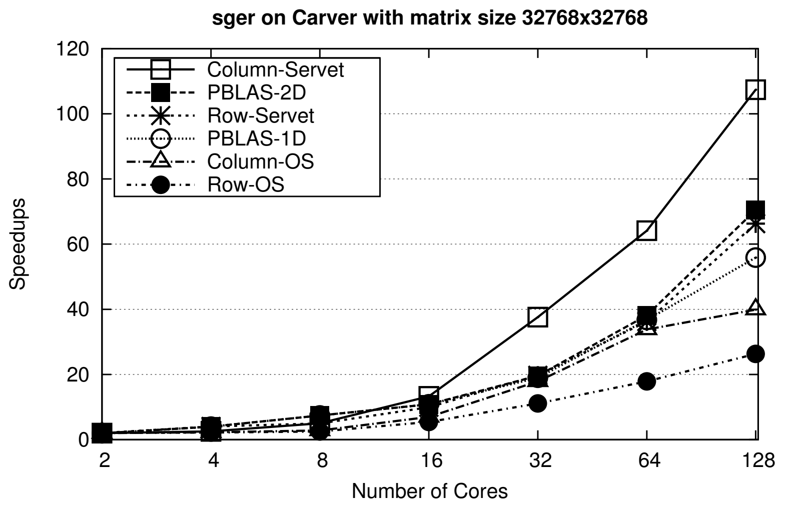


Figure 4.3: Weak scaling speedups of the single precision matrix-vector product (*sgemv*)

uct (*sger*) for strong scaling. The main difference with the matrix-vector product (*sgemv*) is that the distribution by columns does not need a final reduction. As can be seen in Figures 2.6 and 2.7 both distributions use a similar algorithm, where each thread needs to copy one of the input vectors but it has the corresponding rows or columns of the matrix in local memory. Therefore, the execution times of both versions should be similar. However, on Carver the distribution by columns obtains higher scalability than by rows. The reason is the inclusion of cache access optimizations in the MKL library. In MKL the computation time within each



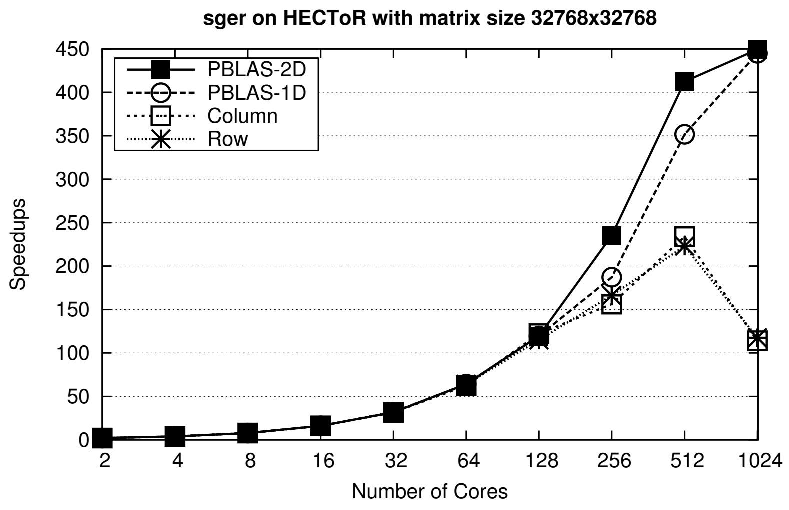


Figure 4.4: Strong scaling speedups of the single precision outer product (*sger*)

UPC thread is minimized thanks to cache reuse of its corresponding submatrix of A when this submatrix has fewer columns than rows (distributions by columns of Figure 2.7). Consequently, the outer product with the column distribution obtains the best performance.

Two other issues must be noted regarding the Carver experiments. On the one hand, the mapping policy provided by Servet improves performance between two and three times with respect to the scheduling used by the OS. On the other hand, UPCBLAS is very competitive compared to PBLAS even for the largest number of

Table 4.6: Strong scaling execution times (in milliseconds) of the single precision outer product (*sger*) on Carver

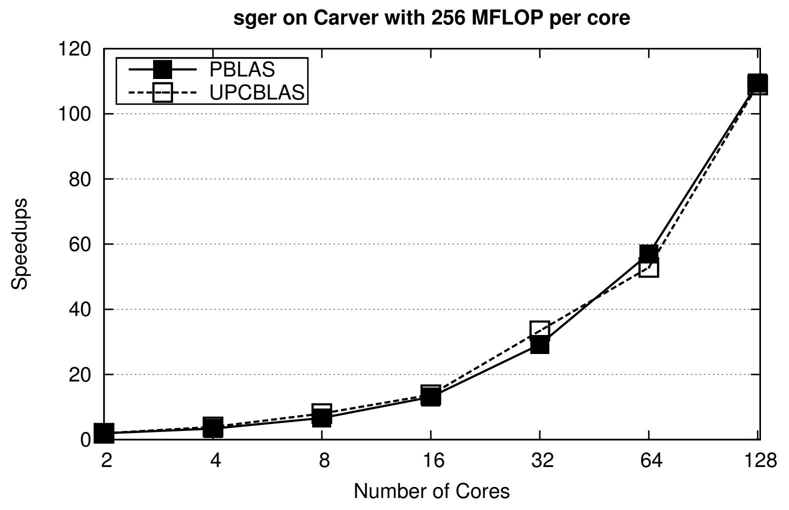
sger with matrix size 32768x32768 (ms)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	OS Map	Servet Map	OS Map	Servet Map	1D	2D
Seq	656.22					
2	323.81	323.81	329.93	327.98	334.80	334.80
4	301.51	162.65	292.23	258.13	166.06	166.06
8	265.20	132.65	235.79	131.11	88.64	88.64
16	121.35	66.12	94.30	49.45	60.56	60.56
32	59.20	33.29	36.71	17.46	34.51	33.50
64	36.69	18.11	19.42	10.23	17.87	17.21
128	24.92	9.90	16.45	6.11	11.75	9.31

Table 4.7: Strong scaling execution times (in milliseconds) of the single precision outer product (*sger*) on HECToR

sger with matrix size 32768x32768 (ms)				
Cores ↓	Row Dist	Column Dist	PBLAS 1D	PBLAS 2D
Seq	907.49			
2	459.44	459.59	453.74	453.74
4	234.20	234.12	235.11	235.11
8	117.31	117.69	116.34	116.34
16	56.85	56.59	56.70	56.70
32	28.21	28.87	28.60	28.60
64	14.23	14.55	14.15	14.15
128	7.89	7.43	7.60	7.60
256	5.46	5.81	4.85	3.86
512	4.07	3.88	2.58	2.20
1024	7.73	7.96	2.04	2.02

cores. In this case, the UPC version with both distributions obtains better performance than the `PBLAS-1D` outer product, and the UPC column distribution even outperforms `PBLAS-2D`.

As the outer product of LibSci is not as optimized as the MKL one, UPCBLAS performance is similar to PBLAS up to 128 cores on HECToR, being less efficient for a larger number of cores. According to the weak scaling results shown in Figure 4.5,



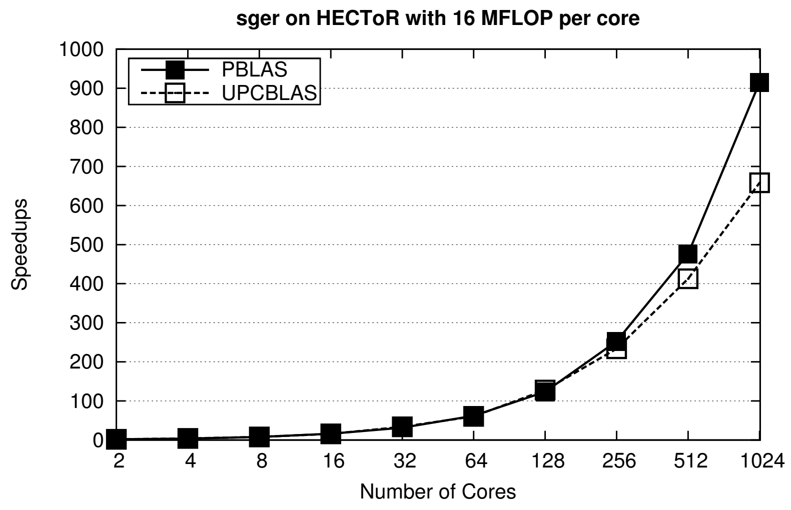
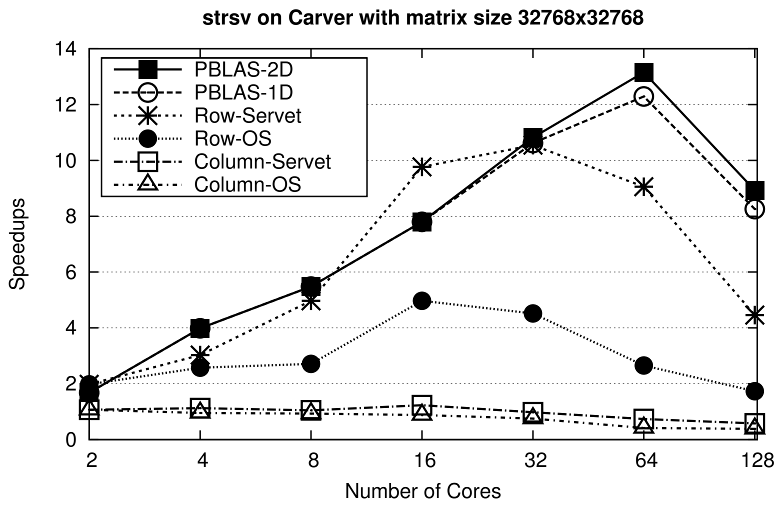


Figure 4.5: Weak scaling speedups of the single precision outer product (*sger*)

it can be assumed that for experiments with larger workload UPCBLAS *sger* would also be competitive for a large number of cores.

The strong scaling experimental results for the last BLAS2 routine, the triangular solver, are shown in Figure 4.6 and Tables 4.8 and 4.9. In this case, besides the choice between row and column distribution, the block size has a great impact on the performance of the parallel solver, as can be inferred from Algorithm 2.1 in Chapter 2. The more blocks the matrix is divided into, the more computations can be simultaneously performed, but the more synchronizations are needed as well.



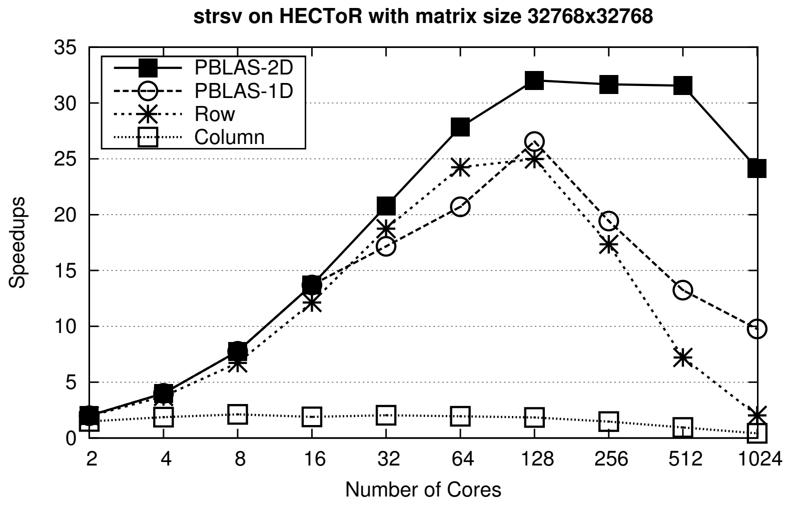


Figure 4.6: Strong scaling speedups of the single precision BLAS2 triangular solver (*strsv*)

Thus, it is necessary to find a good trade-off between parallelism and synchronization overhead. Only the best execution times and speedups are shown for both UPCBLAS and PBLAS by selecting the most appropriate block size.

Two issues previously observed in the matrix-vector and outer products are repeated for the triangular solver. On the one hand, the application of the mapping automatically determined by Servet optimizes the performance and scalability of the UPCBLAS routine on Carver, regardless of the chosen distribution. On the other

Table 4.8: Strong scaling execution times (in milliseconds) of the single precision BLAS2 triangular solver (*strsv*) on Carver

strsv with matrix size 32768x32768 (ms)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	OS Map	Servet Map	OS Map	Servet Map	1D	2D
Seq	235.58					
2	119.09	119.09	220.26	220.26	140.14	140.14
4	91.46	77.84	247.79	209.54	59.14	59.14
8	86.81	47.38	252.17	224.47	42.97	42.97
16	47.40	24.12	267.28	191.21	30.22	30.22
32	52.12	22.34	312.75	241.16	22.18	21.77
64	88.75	26.02	569.43	319.95	19.17	17.91
128	135.80	52.82	613.63	407.12	28.54	26.38

Table 4.9: Strong scaling execution times (in milliseconds) of the single precision BLAS2 triangular solver (*strsv*) on HECToR

strsv with matrix size 32768x32768 (ms)				
Cores ↓	Row Dist	Column Dist	PBLAS-1D	PBLAS-2D
Seq	397.49			
2	202.60	268.03	198.11	198.11
4	106.82	212.22	99.36	99.36
8	59.12	187.23	51.22	51.22
16	32.76	209.43	29.02	29.02
32	21.20	194.56	23.16	19.15
64	16.39	204.26	19.20	14.28
128	15.91	215.21	14.97	12.41
256	22.91	269.65	20.46	12.55
512	55.11	421.52	30.03	12.60
1024	196.58	948.80	40.72	16.47

hand, both UPCBLAS and PBLAS obtain higher speedups on HECToR than on Carver, but because the sequential reference time is also higher.

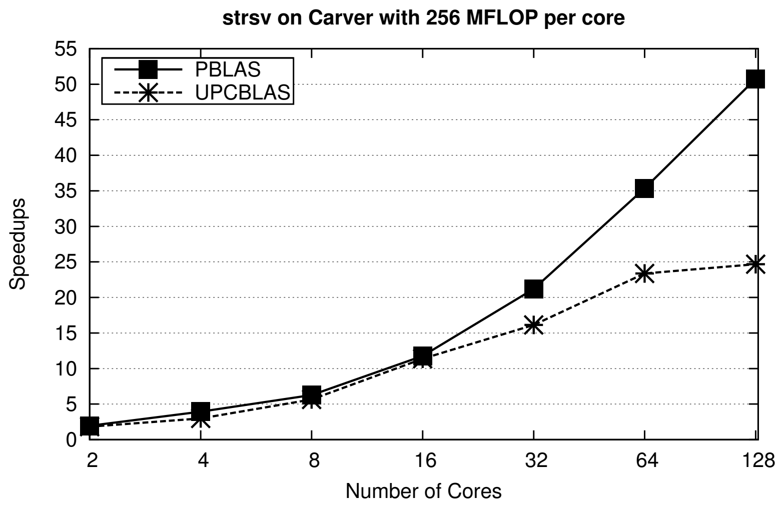
As explained in Section 2.1.2, the only acceptable 1D approach (in terms of performance) for the BLAS2 triangular solver is the distribution by rows. The column distribution was included in UPCBLAS just in case it is adequate for other numerical routines using the same matrix or the matrix was obtained from other applications with this distribution. However, it must not be expected to perform better than the sequential algorithm and thus the row distribution is obviously the best approach to use in UPCBLAS.

The behavior of this routine is very different to the products as it includes internal broadcasts and synchronizations among all threads. Therefore, this is not a scalable routine for strong scaling: the necessary communications and synchronizations involve so much overhead for short execution times that it is not worth it to parallelize it using many cores. This hypothesis is demonstrated by the experimental results, as the speedups are very low not only for UPCBLAS but also for the `PBLAS-2D` algorithm; the PBLAS routine even stops scaling at 64 cores on Carver and at 128 cores on HECToR.

As can be seen in Figure 4.7, better results are obtained for weak scaling as the large workload per core compensates for the additional overhead due to broadcasts and synchronizations. Additional results not shown in the graphs illustrated that, in this scenario, the performance of the UPCBLAS routine with the distribution by rows is very similar to the PBLAS routine with the same 1D distribution. Both approaches (UPCBLAS and PBLAS-1D) scale up to 256 cores on HECToR although their weak scaling speedup is, in general, quite lower than using the best 2D distribution with PBLAS.

4.2.3. BLAS3 Routines

BLAS3 routines are the most interesting ones in terms of parallelization. These routines usually need a large amount of memory (they only work with matrices instead of vectors) and their sequential times are usually high so it is worth it to parallelize them. In fact, most commercial numerical libraries, such as MKL or LibSci, provide multithreaded versions of BLAS3 routines using OpenMP. UPC programmers can take advantage of this underlying parallelization without changing their codes or the calls to UPCBLAS routines, just linking to the adequate version of the library. In this case, instead of creating one UPC thread per core, each UPC thread can be related to several cores that share memory. Thus, the internal numerical computation within each UPC thread is performed in parallel using OpenMP on that group of cores. Three are the main advantages of this two-level parallelism:



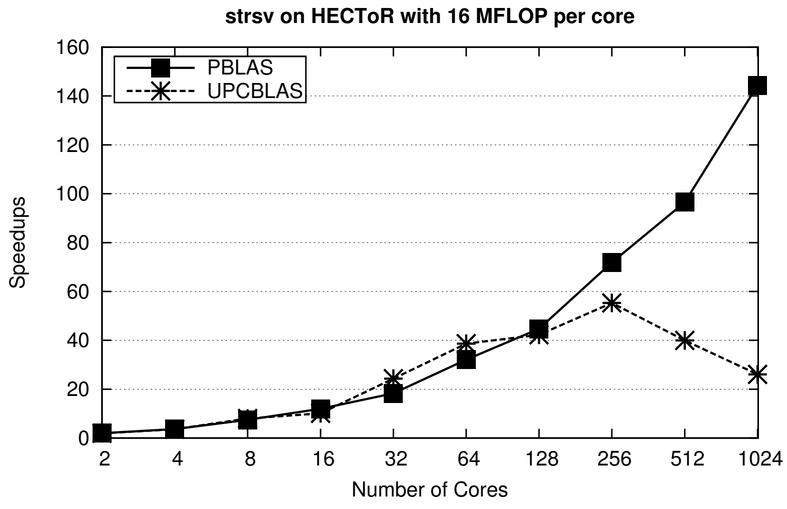


Figure 4.7: Weak scaling speedups of the single precision BLAS2 triangular solver (*strsv*)

- The number of UPC copies is minimized. As there are fewer UPC threads then on-demand copies (see Section 3.7.2) are cheaper, especially on machines such as HECToR where the impact of network contention is very important and concurrent copies or messages involving several cores decrease performance compared to isolated copies.

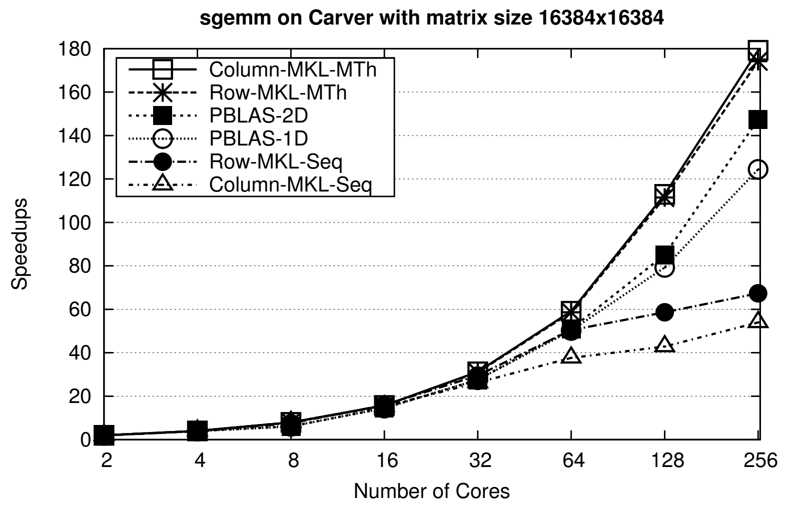
- An internal 2D distribution can be used and thus 2D algorithms (more efficient than 1D ones for many numerical routines) can be applied. For instance, if there are four cores per UPC thread, probably the OpenMP implementation will use a 2x2 grid. Therefore, N being the total number of UPC threads, a 2D approach with a $2N$x2 grid can be employed.

- The internal parallel OpenMP implementation is usually very optimized as the vendor exploits some low-level features of the hardware.

For the multithreaded version, the usage of the highest number of cores that share memory does not always obtain the best performance [26]. In this section results for the UPC BLAS3 routine linked to sequential and multithreaded underlying libraries are shown. After studying the best option with preliminary experiments, the multithreaded versions were executed with 8 OpenMP threads. This means, using all the cores of each node on Carver and of each NUMA region on HECToR. On Carver, experiments linking to sequential MKL were executed using the `SERVET_COMM_PRIOR` policy. Thus, in this case, there is no difference with the automatic mapping provided by the OS.

Figure 4.8 and Tables 4.10 and 4.11 show the speedups and execution times of the single precision matrix-matrix product (*sgemm*) using strong scaling. UPCBLAS results linking to the sequential version of the underlying numerical library are labeled as `MKL-Seq` or `LibSci-Seq` depending on the machine (Carver or HECToR, respectively). Results with two-level parallelism are labeled as `MKL-MTh` and `LibSci-MTh`. The PBLAS results illustrated were obtained always linking to the sequential library as the use of only MPI processes obtains better performance than applying two-level parallelism. The first conclusion that can be made is that the selection of the data distribution (by rows or columns) has no significant influence on the performance of the routine. This is an expected behavior as the algorithms are similar (see Figures 2.9 and 2.10): they perform the on-demand copy of one whole matrix (in the experiments the size of all matrices is the same) and call the *sgemm* routine of the underlying BLAS library. Thus, users can work with the most suitable distribution according to the needs of the other sections of the code.

Additional conclusions depend on the testbed and, especially, on the underlying numerical library. The scalability of all the UPCBLAS versions on Carver is high: all of them scale up to the scenario with the largest number of cores (256). Although the performance of the pure UPC version (i.e. linked to the sequential MKL) is lower



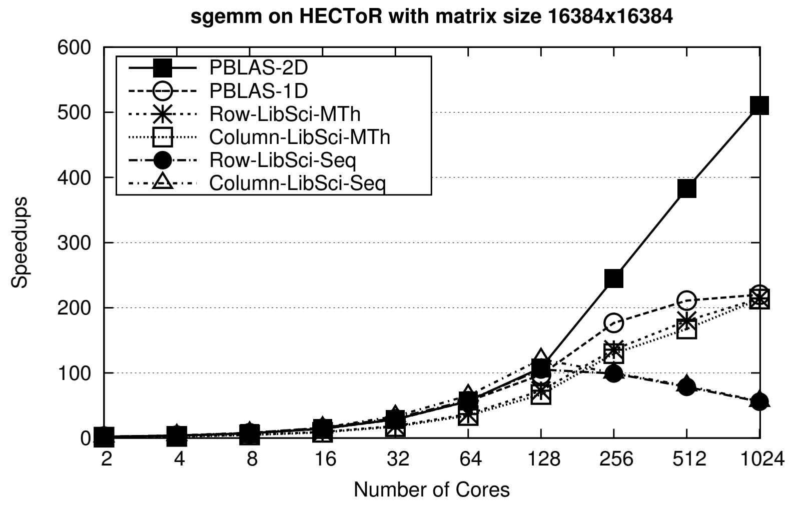


Figure 4.8: Strong scaling speedups of the single precision matrix-matrix product (*sgemm*)

than PBLAS for 128 and 256 cores, UPCBLAS has the opportunity to significantly outperform the PBLAS product when linking to the multithreaded MKL. It must be remarked that the efficiency of this matrix-matrix product is very high with the ease of programming provided by the UPC shared arrays with 1D distributions. Thus, for systems like Carver, the advantage of using the UPCBLAS product is twofold: it is easier to employ and it obtains better performance than PBLAS.

Table 4.10: Strong scaling execution times (in seconds) of the single precision matrix-matrix product (*sgemm*) on Carver

sgemm with matrix size 16384x16384 (s)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	MKL-Seq	MKL-MTh	MKL-Seq	MKL-MTh	1D	2D
Seq	439.16					
2	226.16	222.21	225.74	222.21	222.47	222.47
4	114.13	110.58	110.11	110.58	111.06	111.06
8	58.48	55.39	55.02	55.39	71.96	71.96
16	27.55	27.93	28.85	28.01	30.31	30.31
32	14.89	14.20	16.74	14.06	15.92	15.92
64	8.71	7.49	11.66	7.43	8.74	8.62
128	7.49	3.94	10.25	3.89	5.54	5.16
256	6.52	2.52	8.14	2.45	3.53	2.98

Table 4.11: Strong scaling execution times (in seconds) of the single precision matrix-matrix product (*sgemm*) on HECToR

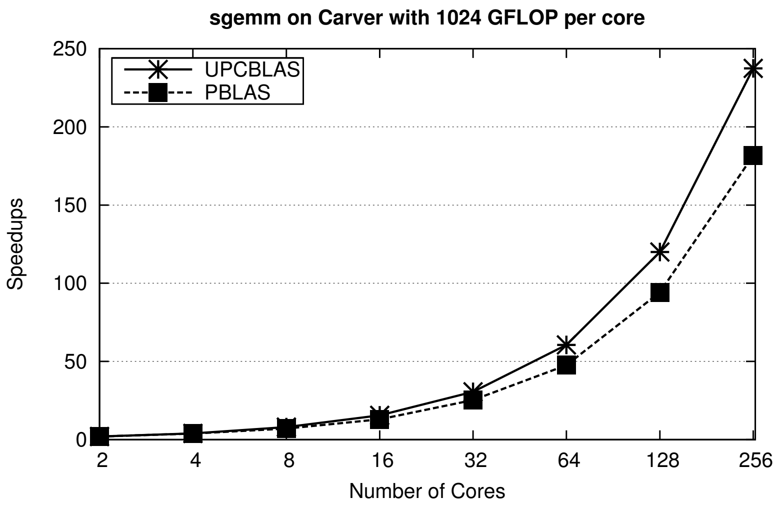
sgemm with matrix size 16384x16384 (s)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	LibSci-Seq	LibSci-MTh	LibSci-Seq	LibSci-MTh	1D	2D
Seq	413.53					
2	211.55	325.64	205.38	325.38	208.75	208.75
4	104.84	163.14	103.08	163.14	114.17	114.17
8	52.60	87.77	51.74	87.77	57.32	56.99
16	26.58	44.12	26.09	46.66	29.07	28.59
32	13.61	22.28	12.42	23.50	14.55	14.40
64	7.13	11.22	6.35	11.98	7.32	7.30
128	3.93	5.69	3.42	6.20	4.26	3.83
256	4.17	3.05	4.14	3.19	2.34	1.69
512	5.27	2.30	5.15	2.47	1.96	1.08
1024	7.42	1.93	7.33	1.94	1.88	0.81

On the Cray testbed (HECToR) the comparison is not as beneficial for UPCBLAS. In this case the `PBLAS-2D` version obtains the best performance. The main reason is that the multithreaded LibSci *sgemm* is not as efficient as MKL (for instance, only 4.93 of speedup for 8 cores). This leads the version linked to the sequential LibSci to obtain better performance than the one linked to the multithreaded LibSci for experiments up to 128 cores. However, it stops scaling for 256 cores because the communication overhead introduced by the on-demand copies (see Section 3.7.2) does not allow the matrix-matrix product to perform in less than 3 seconds. For scenarios with a larger number of cores the usage of the multithreaded LibSci decreases the number of UPC threads and thus minimizes the time needed to perform the on-demand copies. Consequently, UPCBLAS continues scaling at least up to 1024 cores, almost reaching the speedups of `PBLAS-1D`. Besides, looking at the trend of the lines, the UPCBLAS matrix-matrix product linked to the multithreaded LibSci can be assumed to outperform the `PBLAS-1D` routine for a larger number of cores. It must be remarked that the sequential LibSci *sgemm* is very efficient (the sequential time is even lower than MKL on Carver), in contrast to than BLAS1 and BLAS2 routines where MKL outperforms LibSci.

Weak scaling speedups selecting the best distributions for all routines on both systems are shown in Figure 4.9. The conclusions are similar to those of strong scaling results but, as previously seen for BLAS2 routines, speedups are much higher, demonstrating that parallelism is more valuable for large problems. For instance, UPCBLAS *sgemm* achieves an almost ideal speedup on Carver.

The experimental results for the BLAS3 triangular solver *strsm* are shown in Figure 4.10 and Tables 4.12 and 4.13. The conclusions are basically similar on both testbeds. The best performance of the UPCBLAS routine for a large number of cores is achieved when it is linked to multithreaded libraries, either for the row or the column distribution. Comparing both, the efficiency of the algorithm with the column distribution (see Figure 2.11) is much higher than the row distribution explained in Algorithm 2.2. After an analysis of the results, the overhead of the broadcasts in Algorithm 2.2 was determined as the reason of this difference. It must be remarked that the value of the `block_size` parameter has not a significant influence on performance for the column distribution. However, in the row distribution it has a great impact on the speedups of the parallel solver, similar to the BLAS2 counterpart. The more blocks the matrix is divided into, the more computations can be simultaneously performed, but the more synchronizations are needed.

Comparing with the PBLAS routines, it can be asserted that the best UPCBLAS *strsm* presents high scalability: it obtains similar scalability than `PBLAS-2D` on Carver. Furthermore, the column distribution is competitive in all cases compared to



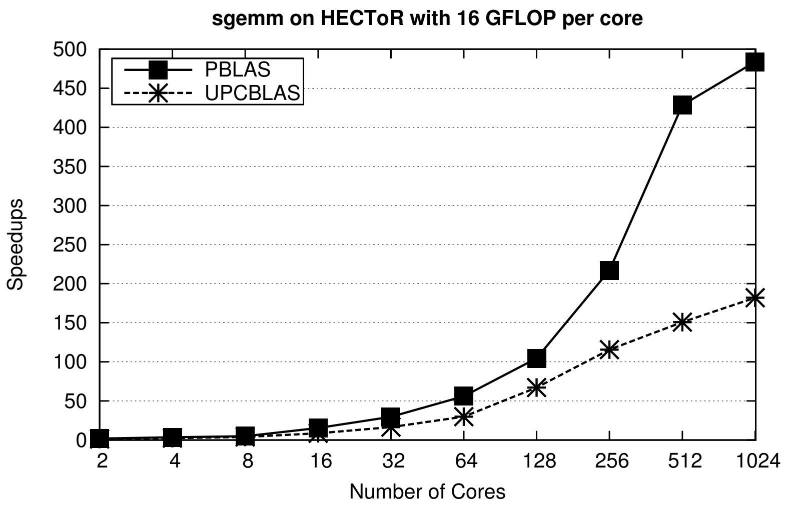
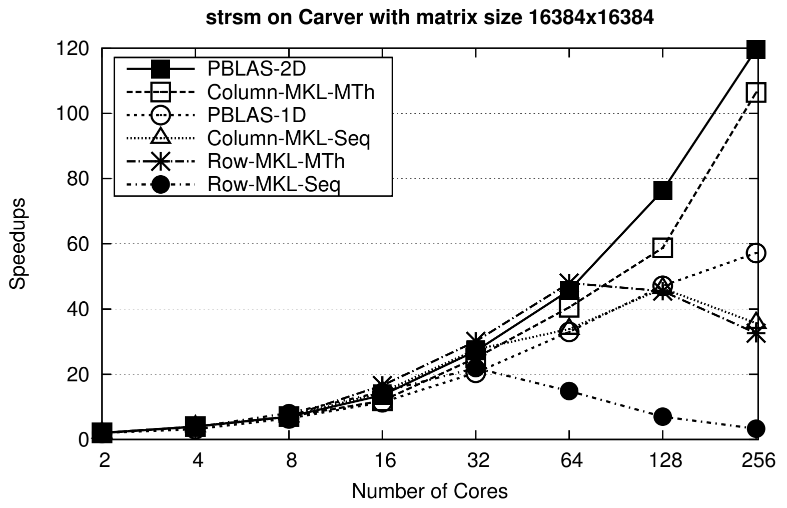


Figure 4.9: Weak scaling speedups of the single precision matrix-matrix product (*sgemm*)

`PBLAS-1D` and even obtains better performance for 1024 cores on HECToR. The difference with the 2D version is more significant on this machine as the multithreaded LibSci routine is not very efficient: it only obtains a speedup of 4.04 for each NUMA region (8 cores). On this machine the best UPCBLAS performance is more similar to the `PBLAS-1D` than to `PBLAS-2D`. However, the `PBLAS-1D` solver stops scaling at 512 cores while the UPCBLAS routine continues increasing its speedups, thus the difference should increase for a larger number of cores. Finally, these conclusions are similar for weak scaling results, as can be seen in Figure 4.11.



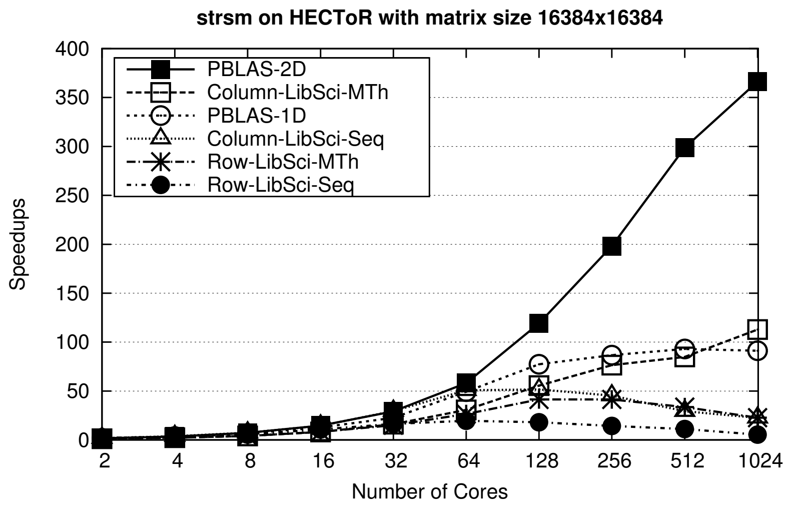


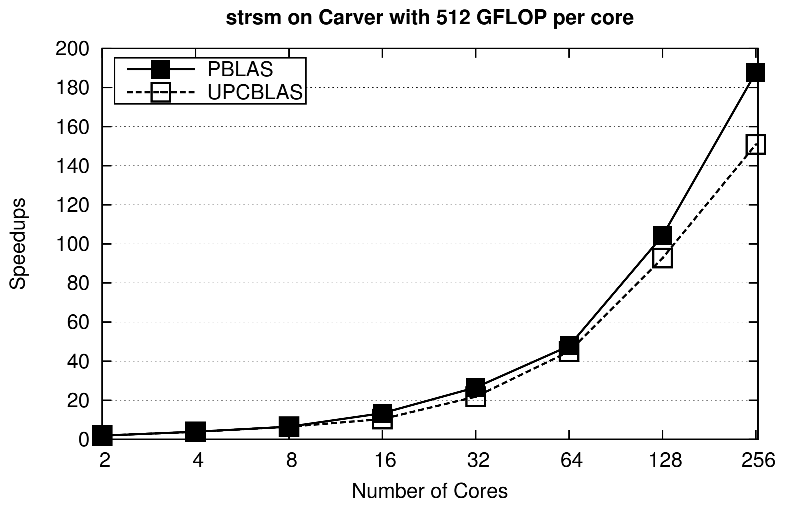
Figure 4.10: Strong scaling speedups of the single precision BLAS3 triangular solver (*strsm*)

Table 4.12: Strong scaling execution times (in seconds) of the single precision BLAS3 triangular solver (*strsm*) on Carver

strsm with matrix size 16384x16384 (s)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	MKL-Seq	MKL-MTh	MKL-Seq	MKL-MTh	1D	2D
Seq	259.64					
2	128.55	128.40	130.19	128.40	129.29	129.29
4	67.43	65.32	65.17	65.32	81.75	65.81
8	32.16	36.87	37.45	36.87	40.12	37.16
16	17.73	15.68	17.94	21.86	22.71	19.09
32	11.86	8.63	9.41	10.39	12.69	9.58
64	17.48	5.41	7.67	6.41	7.87	5.68
128	37.10	5.71	5.60	4.42	5.51	3.40
256	79.26	7.97	7.28	2.44	4.54	2.17

Table 4.13: Strong scaling execution times (in seconds) of the single precision BLAS3 triangular solver (*strsm*) on HECToR

strsm with matrix size 16384x16384 (s)						
	Row Dist		Column Dist		PBLAS	
Cores ↓	LibSci-Seq	LibSci-MTh	LibSci-Seq	LibSci-MTh	1D	2D
Seq	194.17					
2	103.21	170.16	104.20	170.16	113.29	113.29
4	56.94	86.32	51.96	86.32	56.38	56.38
8	32.90	48.06	26.20	48.06	29.11	26.78
16	18.10	23.55	13.38	23.88	15.23	13.31
32	11.80	12.86	6.59	12.09	8.61	6.64
64	9.96	7.32	3.83	6.34	3.94	3.34
128	10.75	4.69	3.76	3.50	2.51	1.63
256	13.59	4.69	4.29	2.54	2.24	0.98
512	17.55	5.80	6.63	2.29	2.09	0.65
1024	35.78	8.52	8.63	1.72	2.13	0.53



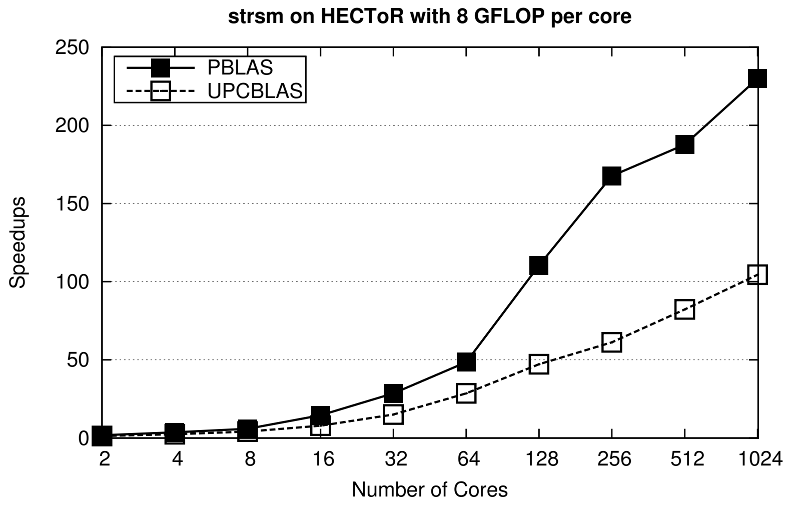


Figure 4.11: Weak scaling speedups of the single precision BLAS3 triangular solver (*strsm*)

4.3. Benchmarking of UPCBLAS within More Complex Routines

The main goal of a numerical library is to simplify the development of more complex codes. This issue is even more important for parallel computing, as programming is more difficult and the increase of productivity can be more significant. For instance, LAPACK [69] uses BLAS routines [6, 33], and PBLAS [21, 83] is part of the ScaLAPACK library [105]. Thus, the UPC versions of some ScaLAPACK routines have been implemented using the UPCBLAS routines to study the convenience of its design and the performance achieved. Specifically, the UPCBLAS library has been used to solve linear systems of equations through Cholesky and LU factorizations. The developed codes have been experimentally evaluated on Carver using up to 256 cores and compared to MPI counterparts using ScaLAPACK. Similar experiments could not be performed on HECToR due to restrictions in the available execution time on this machine.

Parallel solvers are present in many parallel numerical applications and they have been traditionally developed using MPI. This section shows that UPCBLAS can be considered a good alternative to MPI-based libraries for increasing the productivity of numerical application developers.

4.3.1. Cholesky Solver

The Cholesky factorization is mainly used for the numerical solution of linear systems $A * X = B$ when A is symmetric and positive definite. The system can be solved by first computing the Cholesky factorization $A = LL^T$ (L being a lower triangular matrix with strictly positive diagonal entries), then solving $L * Y = B$ for Y, and finally solving $L^T * X = Y$ for X. A similar approach is applied if the system has the form $X * A = B$.

Two different algorithms by blocks have been studied to implement the Cholesky solver. Both algorithms are very adequate for parallel numerical codes as they are based on BLAS3 routines, which obtain good scalability. As UPCBLAS is limited to the distributions available for shared arrays in UPC which, up to now, are 1D, users can choose between the block-cyclic distribution by rows or by columns. For the sake of simplicity, all the algorithms in this section will only be explained for a block-cyclic distribution by rows, such as the one shown in Figure 4.12, where A_{ij} are submatrices. The column distribution version can be easily inferred.

As UPCBLAS is focused on increasing programmability, the syntax of all its routines is very similar to the sequential BLAS counterparts. Furthermore, UPCBLAS routines work with arrays as in the sequential numerical libraries instead of using

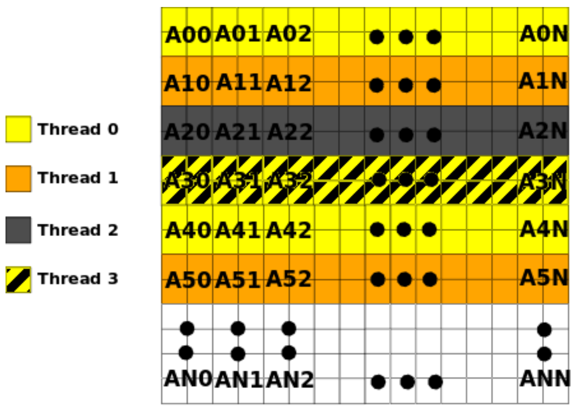


Figure 4.12: Example of input matrix distributed by rows in a block-cyclic way

ad hoc distributed data structures as in the MPI-based ones. Therefore, the first approach for the Cholesky factorization using UPCBLAS is derived from the sequential algorithm available in the LAPACK library. Algorithm 4.1 shows this approach, which is based on the matrix-matrix product of general matrices (*gemm* routine). The input matrix A is distributed in NB blocks of rows. The input/output matrix B (X overwrites B) does not have to follow the same distribution. Taking into account the experimental results obtained for the matrix-matrix product (shown in Section 4.2.3), B is distributed by columns if the system is $A * X = B$, and by rows if it is $X * A = B$, in order to obtain the best performance.

The loop of Algorithm 4.1 computes the Cholesky factorization of matrix A and it is parallelized using UPCBLAS to perform the matrix-matrix product (*gemm*) and the triangular solver (*trsm*). In addition, there is one thread in each iteration that must perform some additional computations only accessing data stored in its local memory: the *syrk* routine (product of a symmetric matrix by its transpose) and the sequential Cholesky factorization. The *syrk* routine is performed by calling a sequential BLAS routine. However, no routine could be used to perform the sequential Cholesky factorization of a block. Although LAPACK has a routine to perform this factorization, it only works if matrices have their elements ordered in a column-wise way, the common format in Fortran. As UPCBLAS follows the UPC and ANSI C format (row ordering of elements), transposing the matrix in each block is the only way to use the LAPACK interface, which was obviously discarded due to its high overhead. An own row-wise C routine for the sequential

Cholesky factorization was therefore implemented. If fact, the employed routine is a multithreaded version that uses OpenMP directives: note that although these operations can be multithreaded they are considered as sequential from the UPC point of view, as they are performed only by one UPC thread.

Algorithm 4.1 Algorithm based on parallel *gemm* for the Cholesky solver

```
for i=0;i<NB;i=i+1 do
    if MYTHREAD has affinity to block i then
        $A_{i,i} = A_{i,i} - A_{i,0..i-1} * A^T_{i,0..i-1} \rightarrow syrk$
        $Sequential\ Cholesky\ Factorization\ of\ block\ A_{i,i}$
    end
    $A_{i+1..N,i} = A_{i+1..N,i} - A_{i+1..N,0..i-1} * A^T_{i,0..i-1} \rightarrow gemm$
    $Solve\ Z * A^T_{i,i} = A_{i+1..N,i} \rightarrow trsm$
    $A_{i+1..N,i} = Z$
end
$Solve\ Y * A^T = B \rightarrow trsm$
$Solve\ X * A = Y \rightarrow trsm$
```

After the loop, the lower triangular part of A stores the entries of L, which can be directly used as input of the UPCBLAS triangular solvers to solve the system of equations.

Two data dependencies arise in this algorithm:

- No thread can start the parallel *trsm* in iteration i before the thread with affinity to block $A_{i,i}$ has finished the Cholesky factorization of this block.
- No thread can start the parallel *gemm* in iteration i before the thread with affinity to block $A_{i-1,i-1}$ has finished its part of the parallel *trsm* of the previous iteration.

There are no dependencies between the parallel *gemm* and the sequential computations in each iteration. Thus, synchronizations have been accordingly used in order to parallelize all these computations.

The second approach developed for implementing the Cholesky solver using UPCBLAS is an adaptation of the algorithm used by ScaLAPACK and it is described in Algorithm 4.2. The UPCBLAS *syrk* and *trsm* routines are used in this case for the parallelization. The main advantage of this algorithm is that there are fewer sequential computations than in the previous one (the sequential *syrk* per iteration is avoided). These computations have not disappeared but they are included in the parallel *syrk*.

Algorithm 4.2 Algorithm based on parallel *syrk* for the Cholesky solver

```
for $i=0;i<NB;i=i+1$ do
    if $MYTHREAD$ has affinity to block $i$ then
        Sequential Cholesky Factorization of block $A_{i,i}$
    end
    Solve $Z * A_{i,i}^T = A_{i+1..N,i} \rightarrow trsm$
    $A_{i+1..N,i} = Z$
    $A_{i+1..N,i+1..N} = A_{i+1..N,i+1..N} - A_{i+1..N,i} * A_{i+1..N,i}^T \rightarrow syrk$
end
Solve $Y * A^T = B \rightarrow trsm$
Solve $X * A = Y \rightarrow trsm$
```

Two dependencies arise in this algorithm too:

- No thread can start the parallel *trsm* in iteration i before the thread with affinity to block $A_{i,i}$ has finished the sequential Cholesky factorization of this block.
- No thread can start the parallel *syrk* in iteration i before all threads have finished the parallel triangular solver in that iteration.

The main drawback of this algorithm is that dependencies are stronger and closer than in Algorithm 4.1. Thus, the overhead due to synchronizations increases.

Figure 4.13 and Table 4.14 show a comparison of the speedups and execution times of both UPC Cholesky algorithms as well as the ScaLAPACK routine of the Cholesky solver. This routine, parallelized using MPI, is provided by the MKL library described in Section 4.1. Note that in these weak scaling experiments the execution times in an ideal scenario (perfect scalability) would remain constant regardless of the number of cores. All the speedups are calculated relative to the execution times obtained from the ScaLAPACK library running with only one MPI process (labeled as `Seq` in the table). The UPC results were obtained linking to

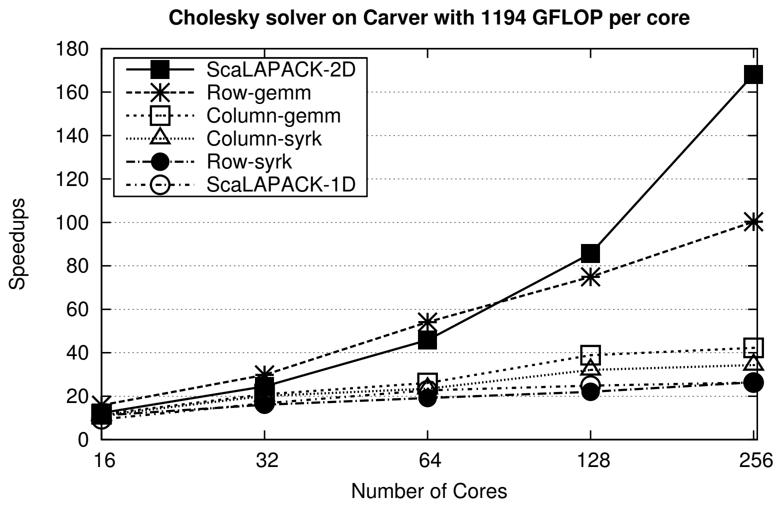


Figure 4.13: Weak scaling speedups of the double precision Cholesky solver on Carver

Table 4.14: Weak scaling execution times (in seconds) of the double precision Cholesky solver on Carver

Cholesky Solver with 1194 GFLOP per core (s)						
	Based on *gemm*		Based on *syrk*		ScaLAPACK	
Cores ↓	Row	Column	Row	Column	1D	2D
Seq	143.43					
16	144.97	197.49	207.31	205.27	245.45	187.18
32	154.17	220.13	286.50	228.46	274.47	187.28
64	169.64	351.98	478.60	391.98	404.97	200.38
128	244.98	472.56	833.37	572.56	738.21	214.43
256	365.90	869.68	1391.89	1069.68	1398.99	218.38

the multithreaded MKL library (using 8 OpenMP threads), which offered the best performance in the experimental results of Section 4.2.3. Consequently, only results from 16 cores (where there are at least two nodes and thus two UPC threads) are shown. Results with row and column distributions for matrix A are shown for both UPC algorithms, where the selection of NB is key to obtain good performance. The more blocks the matrix is divided into, the more computations can be simultaneously performed, but the more synchronizations are needed too. In order to provide a fair comparison, all the experimental results shown were obtained with the best NB for each routine, either for the UPC or the ScaLAPACK routines.

These results indicate that the UPC approach that parallelizes the LAPACK algorithm, based on the parallel *gemm* routine, is better than the adaptation of the ScaLAPACK one, based on the parallel *syrk* routine, due to several reasons:

- The dependencies of the algorithm based on *syrk* are very strong and thus more synchronizations are needed.
- The implementation based on *gemm* has been optimized to minimize the overhead of the sequential computations by overlapping them with other parallel computations. This optimization cannot be included in the *syrk* algorithm because of the dependencies.
- The pattern of the remote accesses within the parallel UPCBLAS *gemm* routine obtains better performance than the pattern within the UPCBLAS *syrk*.

Regarding the best algorithm (the one based on *gemm*) the distribution by rows is better than the distribution by columns. The reason is that the core of the algorithm is performed using the UPCBLAS *gemm* and *trsm* routines. In the distribution by rows these routines follow the behavior described in Figure 2.9 for the matrix product $C = A * B$, with all matrices distributed by rows. As all the elements in the same row have affinity to the same thread, the accesses to one row of B only need one call to the standard UPC function `upc_memget()` per thread. In contrast, in the column distribution case matrix A in Figure 2.10 has the same distribution by columns as matrix B so the elements of each row of A are stored in parts of the shared memory with affinity to different threads. Therefore, several calls to `upc_memget()` per thread with fewer elements per call are necessary to access each row of A. In UPC accessing data using large blocks is much more efficient than splitting them into several smaller accesses.

The ScaLAPACK Cholesky solver outperforms the best UPC version for a large number of cores. The reason is that the 2D distribution is the best choice for this algorithm. Although the best UPC algorithm (the one based on *gemm* by rows) is competitive up to 64 cores, its performance decreases with larger core counts because the blocks become very irregular (few rows and many columns per block). However, this UPC algorithm (including its internal UPCBLAS routines) was demonstrated to be well implemented as it outperforms the `ScaLAPACK-1D` approach and, the most important, UPCBLAS routines are much easier to use.

4.3.2. LU Solver

If the input matrix A does not fulfill the requirements of the Cholesky factorization, the LU decomposition can be used instead. In this case matrix A is decomposed into two matrices L and U, lower and upper triangular, respectively. Thus, A can be exchanged by $L*U$ and the system of equations can be solved in two steps. First, $L*Y=B$ is solved and then $U*X=Y$.

Algorithm 4.3 shows the basic block algorithm based on BLAS3 routines to perform the parallel LU solver. It is derived from the algorithms included in LAPACK and ScaLAPACK. In this case the loop computes the LU factorization, so L and U are stored in the lower and upper triangular parts of A, respectively. Then, the first final *trsm* routine performs the triangular solver with the lower part of the matrix and the second *trsm* uses the upper one.

Algorithm 4.3 Algorithm for the LU solver without pivoting

```
for i=0;i<NB;i=i+1 do
    if MYTHREAD has affinity to block i then
        Sequential LU Factorization of block $A_{i,i..N}$
    end
    Solve $Z * A_{i,i}^T = A_{i+1..N,i} \rightarrow trsm$
    $A_{i+1..N,i} = Z$
    $A_{i+1..N,i+1..N} = A_{i+1..N,i+1..N} - A_{i+1..N,i} * A_{i,i+1..N} \rightarrow gemm$
end
Solve $A * Y = B \rightarrow trsm$
Solve $A * X = Y \rightarrow trsm$
```

This algorithm is mainly based on the UPCBLAS implementations of *trsm* and *gemm*. In addition, similarly to the Cholesky case, a C version (multithreaded using to OpenMP) of the LU factorization had to be developed, as the available libraries could not work with row-wise matrices.

The only dependency among the computations performed by different threads is that no thread can start the parallel *trsm* in iteration i before the thread with affinity to block $A_{i,i}$ has finished the LU factorization of this block. The structure of this algorithm is quite similar to Algorithm 4.2 for the Cholesky solver. However, the LU algorithm should obtain better scalability as no thread needs any remote data of the output of the parallel *trsm* and thus the second dependency present in Algorithm 4.2 (the one between the parallel *trsm* and *syrk* routines in each iteration) is avoided. Moreover, the LU algorithm has been optimized by moving forward the sequential computations of the next iteration, overlapping them with the *gemm* routine.

Depending on the characteristics of the input matrix A, Algorithm 4.3 could lead to incorrect results because of dividing by zero within the sequential LU factorizations. Partial pivoting must be performed in order to avoid this issue, and thus the system has the form $P * L * U * X = P * B$, where P is a permutation matrix with exactly one entry equal to one in each row and column.

Algorithm 4.4 Algorithm for the LU solver with partial pivoting by columns

```
for i=0;i<NB;i=i+1 do
    if MYTHREAD has affinity to block i then
        $P_i = Partial\ Pivoting\ of\ A_{i,i..N}$
        $Swap\ A_{i,i..N}\ according\ to\ P_i$
        $Sequential\ LU\ Factorization\ of\ A_{i,i..N}$
    end
    $Swap\ A_{0..N,i..N}\ according\ to\ P_i$
    $Solve\ Z * A^T_{i,i} = A_{i+1..N,i} \rightarrow trsm$
    $A_{i+1..N,i} = Z$
    $A_{i+1..N,i+1..N} = A_{i+1..N,i+1..N} - A_{i+1..N,i} * A_{i,i+1..N} \rightarrow gemm$
end
$Swap\ B\ according\ to\ P$
$Solve\ A * Y = B \rightarrow trsm$
$Solve\ A * X = Y \rightarrow trsm$
```

The algorithm for solving this system is shown in Algorithm 4.4. If the matrix is distributed by rows, as in the example of Figure 4.12, the pivoting is performed by columns so that all the column swaps can be parallelized. P is implemented as a vector of size N distributed among threads according to NB. It is used to store the information about the columns that must be swapped in each iteration. This information is computed before the LU factorization of the block and it is available to all threads as P is stored in shared memory. Next, all threads have to swap the elements of the rows with affinity to them according to P_i before computing *trsm*

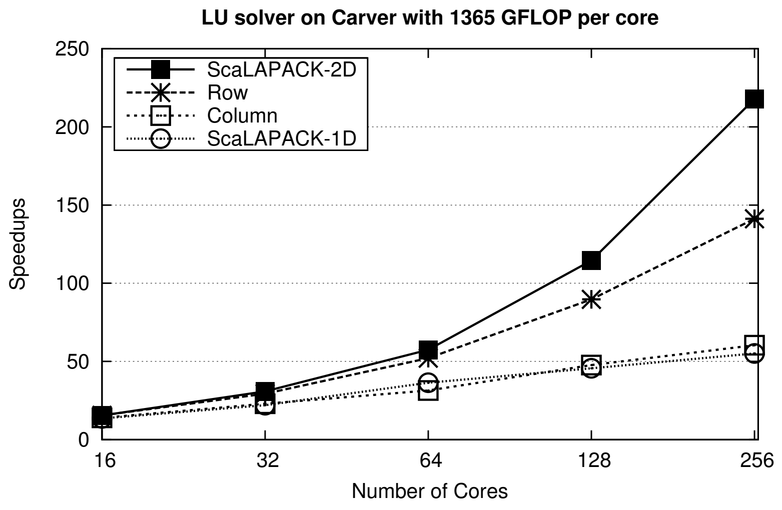
Table 4.15: Weak scaling execution times (in seconds) of the double precision LU solver on Carver

LU Solver with 1365 GFLOP per core (s)				
Cores ↓	Row	Column	ScaLAPACK-1D	ScaLAPACK-2D
Seq	148.10			
16	153.06	170.81	175.08	153.45
32	161.39	206.70	215.14	153.59
64	181.93	301.94	260.52	165.01
128	211.39	397.41	415.57	165.47
256	268.22	628.04	688.22	174.03

in each iteration. A UPC parallel column swap was developed to efficiently perform these computations. Furthermore, in this algorithm the sequential computations of the next iterations cannot be moved forward because the thread involved in these computations needs to finish its part of all the previous parallel *trsm* and *gemm* computations to be sure that the pivoting information is well determined. After the factorization, vector P contains all the pivoting information and columns of matrix B are swapped accordingly in parallel before the final triangular solvers.

Table 4.15 and the top graph of Figure 4.14 show the experimental results for the LU solver without pivoting on Carver. The conclusions are similar to those of the Cholesky algorithm based on *gemm* (see Algorithm 4.1). On the one hand, distributing matrix A by rows obtains better performance than doing it by columns because of the behavior of the UPCBLAS routines; on the other hand, the UPC implementation is much more efficient than `ScaLAPACK-1D` but less efficient than the 2D counterpart.

Finally, weak scaling speedups and execution times of the LU solver with partial pivoting are presented in the bottom graph of Figure 4.14 and in Table 4.16, respectively. In order to show an extreme scenario, matrix A has been selected so that all the columns must be swapped. Regarding the UPC version, only the performance of the distribution by rows is presented as, from the results of the LU solver without pivoting, it can be inferred to be better than the distribution by columns. Comparing to the UPC algorithm without pivoting, it can be seen that pivoting decreases scalability. The reason is not the time needed to perform the swap (only about two seconds per experiment) but the fact that the sequential computations cannot be



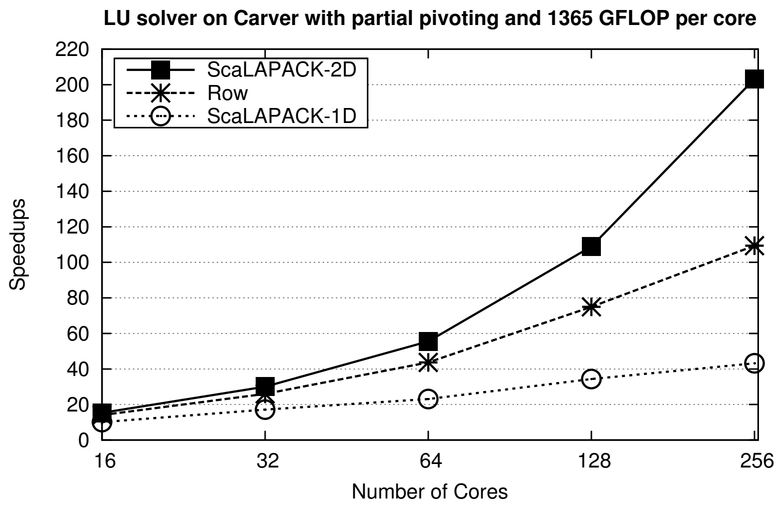


Figure 4.14: Weak scaling speedups of the double precision LU solver with and without partial pivoting on Carver

overlapped with the parallel *gemm* routine of the previous iteration, as explained previously.

The implementations of solvers of equations using the Cholesky and LU factorizations (with and without pivoting) based on UPCBLAS routines have confirmed the ease of use of this library. Thanks to the design features of UPCBLAS (syntax similar to sequential BLAS and use of shared arrays), the usage of its routines has been straightforward, very similar to BLAS routines in LAPACK and much easier than PBLAS routines in ScaLAPACK. Therefore, these solvers illustrate that UPCBLAS routines increase the programmability of parallel numerical codes, producing less error-prone programs and improving the productivity of their users while providing an acceptable performance.

Table 4.16: Weak scaling execution times (in seconds) of the double precision LU solver with partial pivoting on Carver

LU Solver with partial pivoting and 1365 GFLOP per core (s)			
Cores ↓	Row	ScaLAPACK-1D	ScaLAPACK-2D
Seq	150.42		
16	170.65	238.30	157.37
32	185.14	281.26	160.34
64	220.26	416.70	173.34
128	256.89	560.36	176.88
256	351.93	891.54	189.52

Chapter 5

Sparse Numerical Routines in UPC

This chapter analyzes the UPC implementation of SparseBLAS routines and studies the suitability of different sparse storage formats. It begins with a brief summary of the state of the art in Section 5.1. Section 5.2 explains the different sparse formats that will be tested in this chapter. Section 5.3 presents the different possibilities to adapt the design of the UPC dense routines to sparse computations. Section 5.4 shows the algorithms used to implement the sparse matrix-vector and matrix-matrix products, depending on the chosen storage format. Finally, an evaluation of the products using different storage formats is presented in Section 5.5.

5.1. State of the Art

Sparse vectors and matrices are pervasive in many areas, and the efficiency in their processing is critical for the performance of many applications. For instance, sparse matrix-vector and matrix-matrix products represent the main core of many iterative solvers or matrix factorizations that arise in a wide variety of scientific and engineering problems. Due to their importance, several optimization techniques have been proposed for the parallel implementation of sparse products. Williams et al. [121] provide an efficient implementation of the matrix-vector product for multicore systems using the Compressed Sparse Row format by applying thread blocking together with sequential optimizations such as cache blocking, loop optimizations or software memory prefetching. Liu et al. [70] provide another implementation for the Block Sparse Row format using OpenMP. This work also evaluates three different types of load balancing, determining that the non-zero scheduling presented in [66] usually obtains the best performance. A new load balancing method for the sparse matrix-vector product on heterogeneous systems was presented in [65]. Finally, even approaches with autotuning have been developed for this routine [116].

Regarding the sparse matrix-matrix product, Buluç and Gilbert [15] compare different algorithms and data distributions for the multiplication of two sparse matrices. However, this routine is not the same as the one in the SparseBLAS library [99], which multiplies a sparse matrix by a dense one. Nevertheless, none of these works take advantage of the use of PGAS languages. Bell and Nishtala [8] deal with sparse matrices in UPC but restricted to a sparse triangular solver and the Compressed Sparse Row format.

5.2. Sparse Storage Formats

Sparse vectors are usually stored using one array to represent the non-zero elements and another one to indicate their positions. For instance, vector {0.0, 4.3, 0.0, 2.1, 1.9, 0.0} is represented by the `val` array {4.3, 2.1, 1.9} and the `indices` array {1, 3, 4}. All functions that work with sparse vectors use this storage format.

However, several formats have been used in the literature to store sparse matrices. Each of them is adequate for some kind of sparse matrices. The selection of the most suitable storage format is one of the main decisions in order to perform efficient sparse matrix-vector and matrix-matrix products, and this decision can be influenced by the size of the problem, the sparsity pattern of the matrix, the programming language or the architecture of the system. In [71] Luján et al. presented a performance evaluation of different storage formats for the sparse matrix-vector product in Java. This study was complemented in [114] with a similar evaluation using Fortran. Regarding parallel computing, Shahnaz et al. provide in [93] and [94] a comparison of the performance of the sparse matrix-vector product with seven different formats in a small cluster using MPI. Similar studies for GPUs are presented in [9] and [60].

In this chapter the six formats described by Dongarra in [32] are studied, each of them tailored to specific sparsity patterns:

- Coordinate (COO) format: The Coordinate format is the most intuitive, simple and flexible scheme to represent sparse matrices. It consists of three arrays, *values*, *rows* and *columns*, which store the values, row indices and column indices of the non-zero entries, respectively. In most occasions (and always in this work) the non-zero elements of the same row are assumed to be stored contiguously.

- Compressed Sparse Row (CSR) format: The CSR format is probably the most popular sparse matrix representation. It explicitly stores subsequent non-zero values of the rows of the matrix in array *values*. Array *columns* keeps the column indices. A third array *rowPtr* stores, for each row, the index of the entry in the array *columns* which is the first non-zero element of the given row. It has an additional entry with the total number of non-zero elements in the matrix. Therefore, CSR presents a compressed view of Coordinate as the length of *rowPtr* is the total number of rows plus one instead of the total number of non-zero values.

- Block Sparse Row (BSR) format: A variant of CSR is the BSR format, very useful for sparse matrices where the non-zero elements are grouped in blocks. It consists of dividing the matrix in a grid of blocks and keeping, for each block with non-zero entries, its values (including zeros) and the information of the position of the block within the grid according to the CSR scheme. The values are stored consecutively by blocks and, inside them, by rows.

- Compressed Sparse Column (CSC) format: It is similar to CSR, but storing consecutively in array *values* the non-zero elements by columns, using *rows* for the row indices and *columnPtr* for keeping for each column the index of the entry in the array *rows* which is the first non-zero element of the given column.

- Diagonal (DIA) format: Many sparse matrices in scientific computing present their non-zero entries restricted to a small number of diagonals. In order to take advantage of this particular sparsity pattern the Diagonal scheme has been defined. In this case *values* stores consecutively all the elements of the diagonals with any non-zero element. Another array, *distance*, represents, for each stored diagonal, its offset from the main diagonal. Diagonals above and below the main one have positive and negative distance, respectively.

- Skyline (SKY) format: This format has been specifically designed for sparse triangular matrices, frequently used in equations solvers. The concrete storage of the elements depends on whether the matrix is lower or upper triangular. The values of all the entries from the first non-zero element to the diagonal in each row are consecutively stored in array *values* in the lower case. The format is similar for columns in upper triangular matrices. Besides, an additional array *ptr* is necessary. In lower triangular matrices, it keeps for each row the index of the entry of *values* with the first element of this row. In the upper case its meaning is the same but for each column. In both cases an additional entry with the total number of non-zero elements is needed.

5.3. Design of the UPC Sparse Routines

One of the first decisions to take when developing a library is the interface of the routines. In order to support multiple formats for sparse matrices, two options were considered for the UPC sparse routines:

- To provide a different interface for each storage format. Each interface would have the required number of parameters with the most suitable type and name according to the storage format. For instance, the CSR interface would have one array called `val` and two integer arrays called `col_ind` and `row_ptr`.

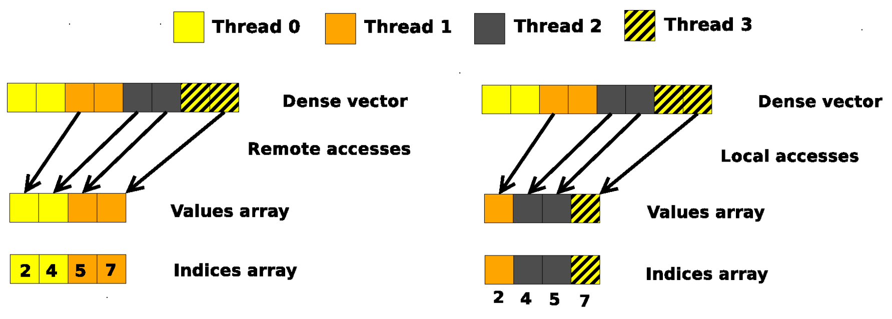
- To provide a unified interface for all storage formats. An enumerated parameter is mandatory to determine which format is used. Parameters used to pass the sparse matrix would have general names and, according to the enumerated value, each of them would have a different meaning. In addition, enough parameters must be available in order to be able to represent all the storage formats, although some of them might be useless for some particular formats.

Looking for ease of use, the first approach has been selected. To distinguish the storage format supported by each routine, their names are suffixed by the abbreviations `_coo`, `_csr`, `_bsr`, `_csc`, `_dia` or `_sky`, according to the formats presented in Section 5.2.

5.3.1. Sparse BLAS1 Routines

The sparse structures present a problem with the affinity that is difficult to solve in UPC. In a sparse vector, non-zero values are usually irregularly distributed, so the affinity with a particular thread is difficult to achieve. Figure 5.1(a) depicts an example that describes the problem for the sparse dot product, which multiplies a dense vector by a sparse one. In this scenario, the shared arrays for both dense and sparse vectors have the same block factor (2 in the example). However, because of the sparsity of the second vector (expressed by the `indices` array), all accesses are remote, which represents significant performance penalties.

In order to avoid remote accesses, the arrays of the sparse structure should be distributed with a non-regular pattern as shown in Figure 5.1(b). In UPC the block factor of a shared array can not be variable, so the data layout depicted in Figure 5.1(b) is not possible using shared arrays.



(a) All arrays are shared (b) Only the dense vector is shared

Figure 5.1: Accesses in the sparse dot product depending on the arrays that are shared

In conclusion, the storage of both dense and sparse vectors in shared arrays is not recommended. Again, similarly to dense routines, programmability is a must. Thus, the approach developed only uses UPC standard structures and provides routines easy to understand and work with. Following these assumptions, the implemented sparse dot product routine has an interface that handles the sparse vector in private memory and internally distributes its data in the best way according to the `block_size` specified by the user, which represents the dense vector distribution. For instance, the syntax of the *susdot* routine (dot product between a sparse and a dense vector of floats) is:

```
int upc_blas_susdot(int block_size, int nz, int thread_src, float x,
                    int *indx, shared float *y, shared float *dst);
```

`x` and `indx` being the source arrays of length `nz` that represent the sparse vector (note that they are stored in private memory); `y` the dense vector; `dst` the pointer to shared memory where the dot product result will be written; `block_size` is the block factor of the dense source vector `y`; and `thread_src` the rank of the thread (0,1. . . `THREADS`-1, `THREADS` being the total number of threads in the UPC execution) where the private input is stored. If `thread_src=THREADS`, the input is replicated in all threads. This function treats pointer `y` as if it had type `shared [block_size] double[indx[nz-1]]`.

5.3.2. Sparse BLAS2 and BLAS3 Routines

As sparse matrices present the same problem as sparse vectors, they are also handled in private memory and internally distributed according to the `block_size` of the dense structures. For instance, the syntax of the function to multiply a sparse matrix in CSR format by a dense vector is:

```
int upc_blas_susmv_csr(UPCBLAS_TRANSPOSE transpose, int m, int n,
                      float alpha, int thread_src, float *val,
                      int *col_ind, int *row_ptr, int block_size,
                    shared float *x, int sec_block_size,
                    shared float *y);
```

val, col_ind and row_ptr being the arrays that represent the sparse matrix; x and y the dense source and result vectors, respectively; m and n the number of rows and columns of the matrix; alpha the scale factor that multiplies the matrix; transpose an enumerated value that indicates whether the sparse matrix is transposed; and thread_src the rank of the thread where the private input is stored. The block factor of vector x is block_size whereas sec_block_size is the block factor of vector y. This function treats pointer x as shared[block_size] double[n] and y as shared[sec_block_size] double[m].

5.3.3. Limitations and Future Directions

The described design has an important drawback: each call to each sparse routine involves an initial distribution of the input sparse vector or matrix from the source thread to the other threads according to block_size. Many calls to the same or different routines with the same distribution are often used when developing more complex algorithms, as could be seen in Sections 4.3.1 and 4.3.2 for the dense counterparts. Thus, in these common scenarios the sparse structure would be continuously redistributed, generating a huge performance overhead.

As it is not possible to have sparse vectors and matrices distributed using shared arrays with a fixed block factor, two options arise to have them distributed before calling the routines, and therefore to avoid the overhead. The first one would be to include shared arrays with non-fixed block factors in the UPC standard. However, this proposal is not one of the priorities of the UPC community. The second option is the design of data structures that represent distributed sparse vectors and matrices, similar to the PETSc ones [85]. The development of these structures must be considered as future work, also providing additional routines for their creation, initialization and deletion that could help users to work with them and thus maintain the high programmability of the UPCBLAS library. As in this case the distribution would be hidden from the user by the data structures and by these additional routines, the storage format could be automatically chosen by the library. Subsequent sections provide an evaluation of different algorithms with different storage formats to perform the sparse matrix-vector and matrix-matrix products that will help to this future automatic choice of the storage format. This is the previous step to the design of those auxiliary structures that could be included in UPCBLAS.

5.4. Implementation of the Sparse Matrix Products

This section analyzes the implementation of the sparse matrix-vector and matrix-matrix products in UPC. The syntax of these products is the same as in the Sparse-BLAS library [99]: $y = \alpha * A * x + y$ and $C = \alpha * A * B + C$, respectively (α being a scalar value, A a sparse matrix, x and y dense vectors, and B and C dense matrices). The goal is to compare the different storage formats to study which is potentially the best one to implement sparse routines in UPC. As explained in the previous section, the direct application of the interface for dense routines to the sparse ones would involve a significant performance overhead. The purpose of this preliminary study is therefore to determine the best algorithm for each storage format, without being limited by any feature of the interface. Thus, not only the sparse but also the dense vectors and matrices are distributed using private memory in order to be able to play with more variable distributions (i.e. with variable block factor).

Figure 5.2 illustrates the approach of the matrix-vector product that distributes the sparse matrix by rows. Each thread calculates a partial result by applying a sequential sparse matrix-vector product with the rows that correspond to it and all the elements of x. Thus, the distribution of y must match the distribution of the matrix so that the partial sums can be performed without remote accesses. In order to perform this distribution the non-zero elements must be consecutively stored by rows in the *values* array so that it can be used in the COO, CSR, BSR and SKY (with lower triangular matrices) formats.

For the CSC and SKY (with upper triangular matrices) formats, where the data in the *values* array are consecutively stored by columns, the use of this row distribution would lead to several data movements, which would represent an important performance overhead. The natural distribution for these formats is by blocks of

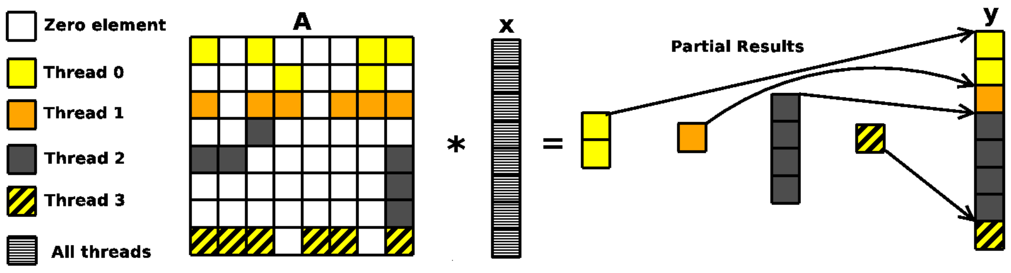


Figure 5.2: Distribution by rows of the sparse matrix-vector product (COO, CSR, BSR and SKY with lower triangular matrices)

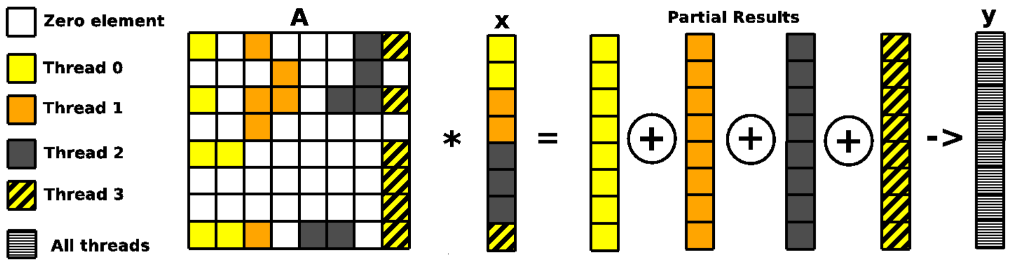


Figure 5.3: Distribution by columns of the sparse matrix-vector product (CSC and SKY with upper triangular matrices)

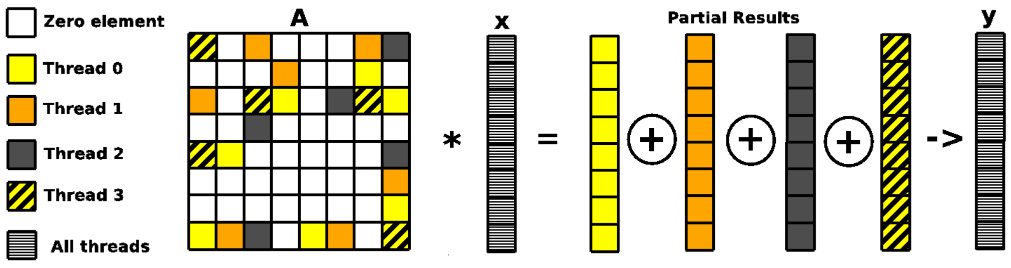


Figure 5.4: Distribution by diagonals of the sparse matrix-vector product (DIA)

columns and with the source vector x distributed according to the size of the blocks in the matrix, as represented in Figure 5.3. Each thread performs a sequential partial sparse matrix-vector product with its local data. Then, in order to compute the i^{th} element of the result, the i^{th} values of all partial results should be added. As in the dense matrix-vector product with column distribution (see Figure 2.5), these sums need reduction operations involving all UPC threads, so performance is usually poor. The approach followed by the DIA format, which can be seen in Figure 5.4, is very similar to the previous one but the sparse matrix is distributed by diagonals and vector `x` is accessed by all threads.

Previous works have pointed out that a key aspect in the performance of the sparse matrix-vector product is the computational load balance [121]. In order to achieve a good load balance the approaches of Figures 5.2 and 5.3 try to evenly distribute the number of non-zero elements per thread, using rows/columns of different size (in the examples, six non-zero elements per thread). Regarding the DIA format, as the number of non-zero elements per diagonal is unknown, the computational load might be unbalanced (in Figure 5.4, seven non-zero elements for threads 0 and 1 and five non-zero elements for threads 2 and 3). Nevertheless, the impact of this drawback is alleviated by using a cyclic distribution which achieves a balanced load distribution in most sparse matrices.

Figure 5.5 shows the algorithm for the matrix-matrix product using an own distribution, which is an adaptation of the matrix-vector algorithm by rows. Each thread needs the whole matrix B and the same rows of C as in A. Nevertheless, the adaptation of the distribution by columns and diagonals employed in the matrix-vector multiplication would eventually involve a significant number of final reductions, leading to a very poor performance. Therefore, the approach illustrated in Figure 5.6 was developed for CSC, DIA and SKY with upper triangular matrices. It is similar to the algorithm for the dense matrix-matrix product with column distribution presented in Section 2.1.3 and illustrated in Figure 2.10. Each thread needs to access the whole sparse matrix but only the same columns of B and C, and the results are calculated always in local memory.

5.5. Performance Evaluation of the Sparse Routines

The evaluation of the UPC sparse products has been conducted on the Finis Terrae supercomputer [40] at the Galicia Supercomputing Center (CESGA). This system consists of 142 HP RX7640 nodes, each of them with 16 IA64 Itanium2 Montvale cores at 1.6Ghz, 128GB of memory and a dual 4X InfiniBand port. As for software, the code was compiled using Berkeley UPC 2.12.1. The intra-node and inter-node communications are performed through shared memory and GASNet over

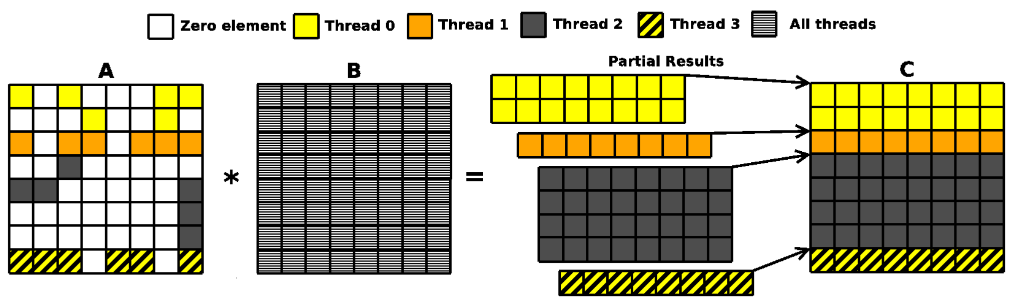


Figure 5.5: Distribution by rows of the sparse matrix-matrix product (COO, CSR, BSR and SKY with lower triangular matrices)

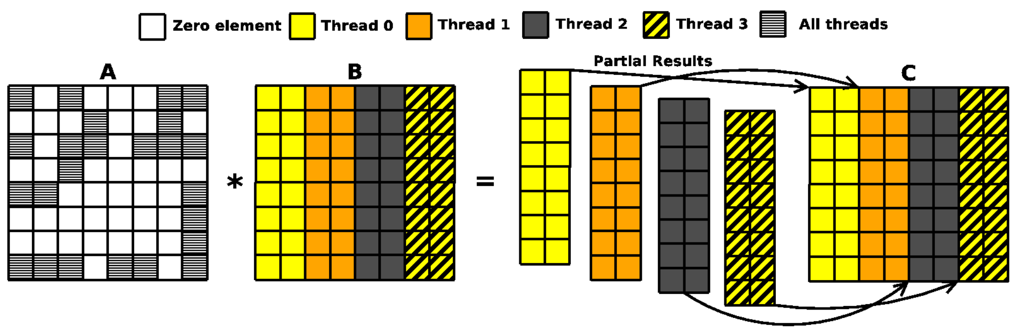
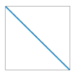
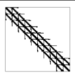
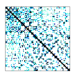
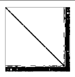
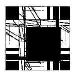
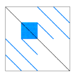
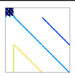


Figure 5.6: Distribution by columns of the sparse matrix-matrix product (CSC, DIA and SKY with upper triangular matrices)

InfiniBand, respectively.

In this evaluation seven representative square matrices, with different sparsity patterns, have been selected from the University of Florida Matrix Collection [107]. Their characteristics are shown in Table 5.1. Larger versions (labeled with `large`) have been obtained by replicating the original matrices, which preserves the sparsity and the pattern of the original ones. The larger versions have been used for the matrix-vector product whereas the original matrices have been used for the matrix-matrix product. The DIA and SKY formats are not appropriate for storing some matrices due to the significant number of zeros that the format would require to store them, and thus these combinations have not been considered. Specifically, `nd3k`, `gupta3` and `pattern1` are not stored using the DIA format while SKY is not appropriate for `nd3k`, `pattern1`, `exdata_1` and `TSOPF`. All the results have been obtained by discarding the overhead of the initial data distribution (for many ap-

Table 5.1: Overview of the sparse square matrices used in the evaluation

Plot	**Name**	**Rows**	**Sparsity**
	nemeth26	9506	0.842%
	nemeth26_large	85554	0.842%
	ramage02	16830	0.509%
	ramage02_large	67320	0.509%
	nd3k	9000	2.03%
	nd3k_large	81000	2.03%
	gupta3	16783	1.658%
	gupta3_large	67132	1.658%
	pattern1	19242	1.26%
	pattern1_large	57726	1.26%
	exdata_1	6001	3.159%
	exdata_1_large	84014	3.159%
	TSOPF	18696	1.258%
	TSOPF_large	56088	1.258%

plications several consecutive products are performed with the same input data distributions).

The top graphs of Figure 5.7 show the speedups of the double precision sparse matrix-vector product using 32 and 64 threads. As expected, the row-based storage formats always significantly outperform column- and diagonal-based ones due to the avoidance of the final reduction operations.

The bottom graphs of Figure 5.7 show the results of the sparse matrix-matrix product. When working with matrices with a quite similar number of non-zero elements per row (for instance `nemeth26`, `ramage02` or `nd3k`) the row distribution obtains better performance than the column one due to the efficiency of the final copies of data to the output array. As can be seen in Figures 5.5 and 5.6, the approach by rows gathers the output data with only one bulk copy per thread, which in UPC is more efficient than using one bulk copy per row and thread, as in the approach by columns. However, if the matrix presents a very irregular sparsity pattern such as `pattern1`, `exdata_1` or `TSOPF`, the distribution by rows tries to balance the number of non-zero elements per thread and this leads to a different number of rows per thread. Therefore, the computational workload of the final sums and data copies is very unbalanced, obtaining less efficiency than distributing the

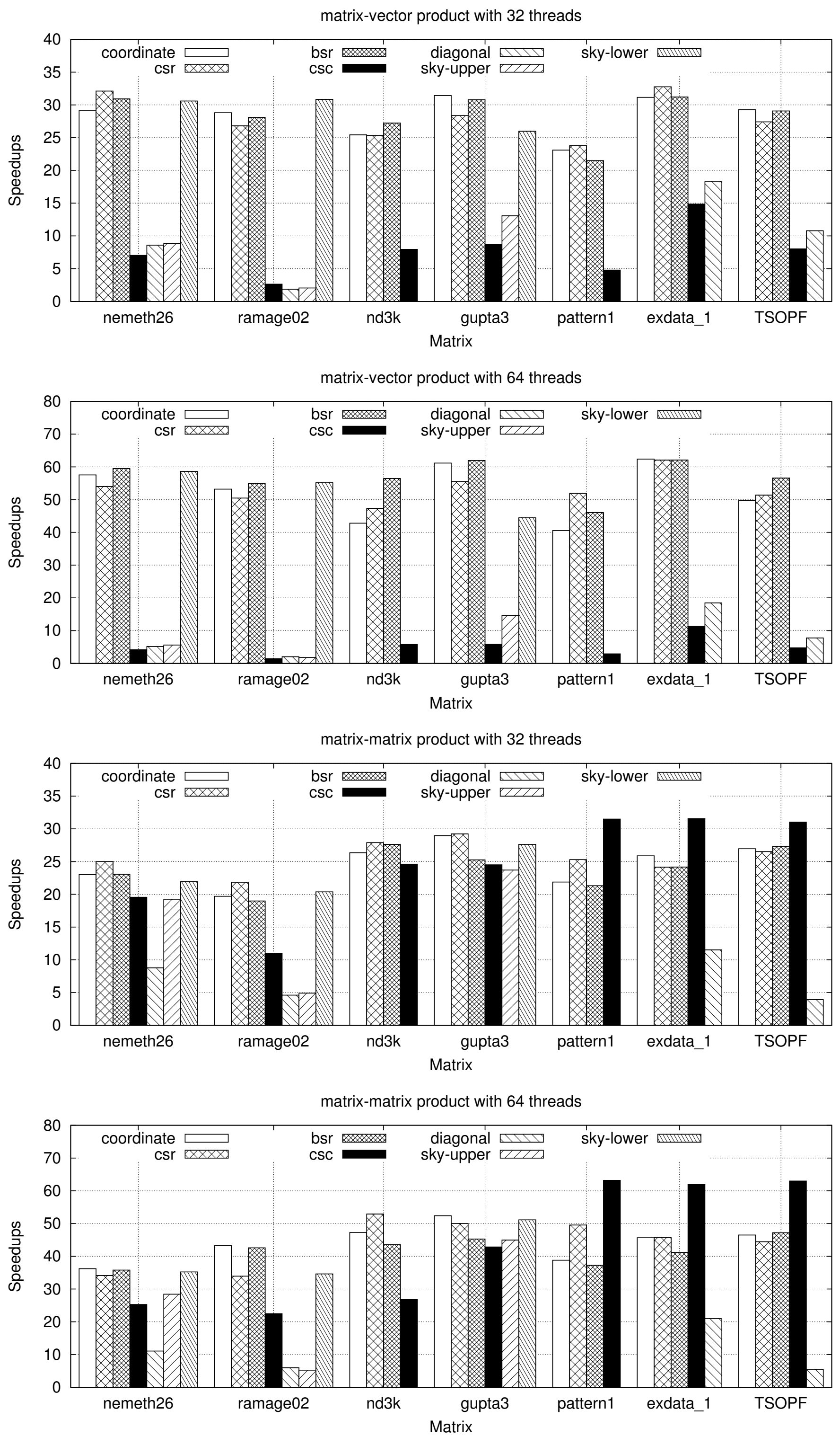


Figure 5.7: Speedups of the sparse matrix-vector and matrix-matrix products for different storage formats

dense matrices by columns. Finally, the poor speedups of the DIA format for this routine is due to the fact that the sequential times are lower than using other formats thanks to the efficient exploitation of the memory hierarchy provided by this format on the evaluated matrices. However, when data in DIA format are distributed among several threads this cache efficiency decreases, showing significantly poorer scalability.

Chapter 6

Conclusions and Future Work

PGAS languages provide programmability and data locality exploitation on shared, distributed and hybrid shared/distributed memory architectures. In fact, PGAS languages such as UPC represent an interesting alternative for programming multicore clusters, where threads running on the same node can access their data efficiently through shared memory, whereas the use of distributed memory improves the scalability of the applications. However, the analysis of the state of the art revealed the lack of available libraries. The development of these libraries could motivate the adoption of this paradigm among the community of parallel programmers. Following this thought, this PhD Thesis, *"UPCBLAS: A Numerical Library for Unified Parallel C with Architecture-Aware Optimizations"*, has developed UPCBLAS, the first parallel numerical library for UPC. It provides parallel versions of the most used BLAS routines that exploit the advantages of the PGAS paradigm. Up to now, in order to use BLAS routines, parallel programmers needed to resort to message-passing-based libraries. With UPCBLAS, UPC programmers can benefit from a portable and efficient PGAS-based library that can also be used as a building block for higher level numerical computations (e.g. factorizations, iterative methods...).

One of the main reasons to use a library is the increase of productivity thanks to avoiding the reimplementation of frequently used codes or kernels, and thus minimizing the time to implement more complex numerical codes. Consequently, providing an easy-to-use interface is key for the adoption of a library. During all the development of UPCBLAS the ease of use has been therefore an important factor in all the design decisions in order to preserve the programmability feature of PGAS languages and reduce the library learning curve. The library works with data distributions provided by the user through the block factor specified in shared arrays. Thus, UPCBLAS is easier and more intuitive to use than message-passing-based numerical libraries (e.g. PBLAS) thanks to directly using vectors and matrices as source and result parameters of the routines (as in sequential BLAS), instead of using complex data structures to handle distributed vectors and matrices. Also, the ease of programming is twofold: on the one hand, the BLAS-like interface facilitates the use of the library to programmers used to sequential BLAS and, on the other hand, the syntax of UPCBLAS is very familiar to UPC programmers as it is similar to that of the UPC collectives library.

Nevertheless, another key aspect for the adoption of a parallel numerical library is the performance of its routines. Several optimization techniques have been applied to improve performance in UPCBLAS. Some examples are privatization of shared pointers, bulk data movements or redesign of some collective operations. Besides, sequential BLAS routines are embedded in the body of the corresponding UPC routines. Using efficient underlying libraries not only improves performance, but it also allows incorporating a two-level parallelism approach by linking multithreaded versions without any change in the UPC code.

Additionally, two optimization techniques that take into account the hardware features of the underlying system were developed: efficient mapping policies and on-demand copies. In order to be portable, these techniques need an automatic mechanism to obtain some hardware parameters. Servet, a benchmark suite that automatically obtains these parameters, is also part of this Thesis. These benchmarks determine the number of cache levels and their sizes, the topology of shared caches, memory access bottlenecks, and communication scalability and overheads. The experimental results proved that the suite provides highly accurate estimates according to the system architecture specifications. Furthermore, the application of Servet is not limited to linear algebra routines or to the UPC language.

The proposed library has been experimentally tested on two multicore clusters with different architectures, underlying numerical libraries and compilers to show the suitability and efficiency of the library for hybrid architectures. The conclusion is that the ease of use of UPCBLAS does not lead to a much worse performance than that of well-established and mature message-passing-based numerical libraries. The productivity of UPCBLAS has also been proven through the development of more complex numerical codes that use its routines. It is clear that UPCBLAS is intuitive enough to save significant time to UPC programmers and it is efficient enough to justify its use.

This Thesis has led to several publications in the area of high performance computing. A preliminary version of the UPCBLAS library was presented in [49]. It provided two options for the interface of the numerical routines and one of them was the basis for the final UPCBLAS design presented in this Thesis. A work addressing UPC triangular solvers was presented in [48], where several algorithms and data distributions were studied and whose results were key to decide the algorithms used for these BLAS2 and BLAS3 routines in UPCBLAS.

The aforementioned Servet tool was presented in [52] as a fully portable suite of benchmarks to obtain the most relevant hardware parameters to support the automatic optimization of applications on multicore clusters. This paper explains in detail all the benchmarks and proves that this tool can extract the features of two representative systems with very different architectures. One of the most interesting optimization techniques that can benefit from Servet is the application of appropriate process mappings. The current version of Servet includes mechanisms to automatically provide, thanks to the extracted hardware parameters, the best process mappings according to the architecture of the system, the number of cores used and the type of code (memory- or communication-intensive). The algorithms to implement these mappings were explained in [53]. This paper also provides an analysis of the impact of the process mapping for the NPB benchmarks on three parallel programming paradigms (MPI, OpenMP and UPC) and using three different platforms.

The core of this Thesis, the UPCBLAS library, was presented in [51]. With a deep basis in the previous papers, this work provides the final interface of the routines, simplifying the one proposed in [49], but at the same time making it more powerful due to including support to work with subvectors and submatrices. This work also introduces the integration of UPCBLAS and Servet to automatically improve the performance of the numerical routines. The poster presented at the

SC'11 conference [50] helped to announce UPCBLAS to the UPC community, and the development of Cholesky and LU solvers were presented in [47] as application examples of the library.

This Thesis was also useful to study other topics related to numerical computation in UPC that could be applied in the near future to new versions of UPCBLAS. On the one hand, an evaluation of different storage formats for sparse products in UPC was presented in [45] and extended in [46]. It will be useful for any future sparse counterpart of UPCBLAS. On the other hand, the impact of avoiding and overlapping communications in the UPC BLAS3 routines (and also in the Cholesky factorization) was analyzed in [44], which could help to optimize future versions of the routines.

This Thesis has demonstrated that the PGAS paradigm is very useful for linear algebra and, although this library is a huge step forward for the increase of productivity of the UPC community, there are still many more numerical routines that programmers could take advantage of. Future work should continue this line to extend the current interface and functionality of UPCBLAS by providing additional functions (e.g. linear solvers, eigenvalue problems or sparse computations).

Finally, future work on Servet will be focused on checking the suite on new available architectures and fixing the inaccuracies of the tool when extracting their hardware parameters. As well, benchmarks to detect network contention will be included in Servet and the process mappings will be adapted to automatically take into account this new parameter.

Bibliography

[1] J. I. Aliaga, F. Almeida, J. M. Badía, S. Barrachina, V. Blanco, M. Castillo, R. Mayo, E. S. Quintana-Ortí, G. Quintana-Ortí, A. Remón, C. Rodríguez, F. Sande, and A. Santos. Towards the Parallelization of GSL. *Journal of Supercomputing*, 48(1):88–114, 2009. pages 13

[2] AMD Core Math Library (ACML). http://developer.amd.com/libraries/acml/pages/default.aspx [Last visited: November 2012]. pages 13

[3] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication Optimizations for Distributed-Memory X10 Programs. In *Proc. 25th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'11)*, Anchorage, AK, USA, 2011. pages 9

[4] C. Barton, C. Casçaval, G. Almási, R. Garg, J. N. Amaral, and M. Farreras. Multidimensional Blocking in UPC. In *Proc. 20th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, volume 5234 of *Lecture Notes in Computer Science*, pages 47–62, Urbana, IL, USA, 2007. pages 18

[5] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterjee, and J. N. Amaral. Shared Memory Programming for Large Scale Machines. In *Proc. 10th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06)*, pages 108–117, Ottawa, Canada, 2006. pages 1

[6] Basic Linear Algebra Subprograms (BLAS) Library. http://www.netlib.org/blas/ [Last visited: November 2012]. pages 12, 78

[7] C. Bell, D. Bonachaea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems using One-sided Communication and Overlap. In *Proc. 20th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'06)*, Rhodes Island, Greece, 2006. pages 1, 7

[8] C. Bell and R. Nishtala. UPC Implementation of the Sparse Triangular Solve and NAS FT. Technical report, University of California, Berkeley, USA, 2004. pages 14, 92

[9] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proc. 21st ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'09)*, Portland, OR, USA, 2009. pages 93

[10] Berkeley UPC Project. http://upc.lbl.gov [Last visited: November 2012]. pages 1, 54

[11] BLAS Technical Forum. http://www.netlib.org/blas/blast-forum/ [Last visited: November 2012]. pages 12, 31

[12] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v2.0. Technical report, Lawrence Berkeley National Laboratory, USA, 2007. pages 12

[13] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P. Wacrenier, and R. Namyst. Structuring the Execution of OpenMP Applications for Multicore Architectures. In *Proc. 24th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'10)*, Atlanta, GA, USA, 2010. pages 2, 35

[14] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proc. 18th Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP'10)*, pages 180–186, Pisa, Italy, 2010. pages 2, 35

[15] A. Buluç and J. R. Gilbert. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *Proc. 37th Intl. Conf. on Parallel Processing (ICPP'08)*, pages 503–510, Portland, OR, USA, 2008. pages 92

[16] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *Proc. 18th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'04)*, Santa Fe, NM, USA, 2004. pages 13

[17] Carver IBM iDataPlex. http://www.nersc.gov/systems/carver-ibm-idataplex/ [Last visited: November 2012]. pages 54

[18] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. pages 8

[19] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proc. 20th ACM Intl. Conf. on Supercomputing (ICS'06)*, pages 353–360, Cairns, Australia, 2006. pages 2, 35, 45

[20] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proc. 17th ACM Intl. Conf. on Supercomputing (ICS'03)*, pages 63–73, San Francisco, CA, USA, 2003. pages 30

[21] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Proc. 2nd Intl. Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science (PARA'95)*, volume 1041 of *Lecture Notes in Computer Science*, pages 107–114, Lyngby, Denmark, 1995. pages 13, 55, 78

[22] Co-Array Fortran. http://www.co-array.org/ [Last visited: November 2012]. pages 1, 7

[23] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran Performance and Potential: An NPB Experimental Study. In *Proc. 16th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, volume 2958 of *Lecture Notes in Computer Science*, pages 177–193, Austin, TX, USA, 2003. pages 8

[24] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 36–47, Chicago, IL, USA, 2005. pages 7, 8

[25] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-array Fortran on Hardware Shared Memory Platforms. In *Proc. 17th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, volume 3605 of *Lecture Notes in Computer Science*, pages 332–347, West Lafayette, IN, USA, 2004. pages 8

[26] J. Cuenca, L. P. García, and D. Giménez. Improving Linear Algebra Computation on NUMA Platforms through Auto-tuned Nested Parallelism. In *Proc. 20th Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP'12)*, pages 66–73, Munich, Germany, 2012. pages 71

[27] J. Cuenca, D. Giménez, and J. González. Towards the Design of an Automatically Tuned Linear Algebra Library. In *Proc. 10th Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP'02)*, pages 201–208, Canary Islands, Spain, 2002. pages 34

[28] J. Cuenca, D. Giménez, J. González, and K. Roche. Automatic Optimisation of Parallel Linear Algebra Routines in Systems with Variable Load. In *Proc. 11th Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP'03)*, pages 409–416, Genova, Italy, 2003. pages 34

[29] K. Datta, D. Bonachea, and K. Yelick. Titanium Performance and Potential: An NPB Experimental Study. In *Proc. 18th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, volume 4339 of *Lecture Notes in Computer Science*, pages 200–214, New Orleans, LA, USA, 2005. pages 8

[30] J. Dinan, P. Balaji, J. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS Model using MPI One-sided Communication. In *Proc. 26th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'12)*, Boston, MA, USA, 2012. pages 7

[31] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proc. 7th ACM Intl. Conf. on Computing Frontiers (CF'10)*, pages 177–186, Cagliari, Italy, 2010. pages 2

[32] J. J. Dongarra. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, chapter 10. SIAM, 2000. pages 93

[33] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988. pages 12, 78

[34] L. A. Drummond, V. G. Ibarra, V. Migallón, and J. Penadés. Interfaces for Parallel Numerical Linear Algebra Libraries in High Level Languages. *Advances in Engineering Software*, 40(8):652–658, 2009. pages 13

[35] A. X. Duchateau, A. Sidelnik, M. J. Garzarán, and D. A. Padua. P-Ray: A Software Suite for Multi-core Architecture Characterization. In *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, volume 5335 of *Lecture Notes in Computer Science*, pages 187–201, Edmonton, Canada, 2008. pages 36

[36] I. S. Duff, M. A. Heroux, and R. Pozo. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Transactions on Mathematical Software*, 28(2):107–114, 2002. pages 12

[37] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Proc. 14th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'02)*, Baltimore, MD, USA, 2002. pages 1, 30

[38] A. Faraj, S. Kumar, B. Smith, A. R. Mamidala, J. A. Gunnels, and P. Heidelberger. MPI Collective Communications on the Blue Gene/P Supercomputer: Algorithms and Optimizations. In *Proc. 23rd ACM Intl. Conf. on Supercomputing (ICS'09)*, pages 489–490, Yorktown Heights, NY, USA, 2009. pages 2, 34

[39] A. Faraj and X. Yuan. Automatic Generation and Tuning of MPI Collective Communication Routines. In *Proc. 19th ACM Intl. Conf. on Supercomputing (ICS'05)*, pages 393–402, Cambridge, MA, USA, 2005. pages 2, 34

[40] Finis Terrae Supercomputer. https://www.cesga.es/en/infraestructuras/computacion/finisterrae [Last visited: November 2012]. pages 55, 100

[41] B. B. Fraguela, Y. Voronenko, and M. Püschel. Automatic Tuning of Discrete Fourier Transforms Driven by Analytical Modeling. In *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 271–280, Raleigh, NC, USA, 2009. pages 34, 38

[42] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005. pages 34

[43] GCC Unified Parallel C. http://www.gccupc.org/ [Last visited: November 2012]. pages 1

[44] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *Proc. 24th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, 2012. pages xvi, 2, 16, 108

[45] J. González-Domínguez, O. García-López, G. L. Taboada, M. J. Martín, and J. Touriño. SparseBLAS Products in UPC: An Evaluation of Storage Formats. In *Proc. 11th Intl. Conf. on Computational and Mathematical Methods in Science and Engineering (CMMSE'11)*, pages 605–617, Benidorm, Spain, 2011. pages xvi, 2, 108

[46] J. González-Domínguez, O. García-López, G. L. Taboada, M. J. Martín, and J. Touriño. Performance Evaluation of Sparse Matrix Products in UPC. *The Journal of Supercomputing*, 2012, In Press. pages xvi, 2, 108

[47] J. González-Domínguez, O. A. Marques, M. J. Martín, G. L. Taboada, and J. Touriño. Design and Performance Issues of Cholesky and LU Solvers using UPCBLAS. In *Proc. 10th IEEE Intl. Symp. on Parallel and Distributed*

Processing with Applications (ISPA'12), pages 40–47, Leganés, Spain, 2012. pages xvi, 2, 108

[48] J. González-Domínguez, M. J. Martín, G. L. Taboada, and J. Touriño. Dense Triangular Solvers on Multicore Clusters using UPC. In *Proc. 11th Intl. Conf. on Computational Science (ICCS'11)*, Singapore, 2011. pages xv, 2, 107

[49] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, and A. Gómez. A Parallel Numerical Library for UPC. In *Proc. 15th Intl. European Conf. on Parallel and Distributed Computing (Euro-Par'09)*, volume 5704 of *Lecture Notes in Computer Science*, pages 630–641, Delft, The Netherlands, 2009. pages xv, 2, 107

[50] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan. A Library for Parallel Numerical Computation in UPC (Poster). In *Proc. 23rd ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'11)*, Seattle, WA, USA, 2011. pages xv, 2, 108

[51] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan. UPCBLAS: A Library for Parallel Matrix Computations in Unified Parallel C. *Concurrency and Computation: Practice and Experience*, 24(14):1645–1667, 2012. pages xv, xvi, 2, 55, 107

[52] J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In *Proc. 24th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'10)*, Atlanta, GA, USA, 2010. pages xvi, 2, 107

[53] J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño. Automatic Mapping of Parallel Applications on Multicore Architectures using the Servet Benchmark Suite. *Computers and Electrical Engineering*, 32(2):258–269, 2012. pages xvi, 2, 49, 107

[54] GSL - GNU Scientific Library. http://www.gnu.org/software/gsl/ [Last visited: November 2012]. pages 13

[55] HECToR: UK National Supercomputing Service. https://www.hector.ac.uk/ [Last visited: November 2012]. pages 54

[56] High Productivity Computing Systems (HPCS). http://www.highproductivity.org/ [Last visited: November 2012]. pages 8

[57] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *Proc. 25th ACM Intl. Conf. on Supercomputing (ICS'11)*, pages 75–84, Tucson, AZ, USA, 2011. pages 2, 35

[58] HP Unified Parallel C. http://www.hp.com/go/upc/ [Last visited: November 2012]. pages 1

[59] HP's Mathematical Software Library (MLIB). http://www.hp.com/go/mlib [Last visited: November 2012]. pages 13

[60] M. R. Hugues and S. G. Petiton. Sparse Matrix Formats Evaluation and Optimization on a GPU. In *Proc. 12th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'10)*, pages 122–129, Melbourne, Australia, 2010. pages 93

[61] P. Husbands and K. Yelick. Multi-threading and One-sided Communication in Parallel LU Factorization. In *Proc. 19th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'07)*, Reno, NV, USA, 2007. pages 14

[62] Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/ [Last visited: November 2012]. pages 13, 54

[63] G. Jin, L. Adhianto, J. Mellor-Crummey, W. N. Scherer, and C. Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *Proc. 25th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'11)*, Anchorage, AK, USA, 2011. pages 8

[64] H. Jin, D. C. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. M. Chapman. High Performance Computing using MPI and OpenMP on Multi-Core Parallel Systems. *Parallel Computing*, 37(9):562–575, 2011. pages 6

[65] C. D. Jiogo, P. Manneback, and P. Kuonen. Well Balanced Sparse Matrix-Vector Multiplication on a Parallel Heterogeneous System. In *Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006. pages 92

[66] K. Kourtis, G. I. Goumas, and N. Koziris. Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication using Index and Value Compression. In *Proc. 37th Intl. Conf. on Parallel Processing (ICPP'08)*, pages 511–519, Portland, OR, USA, 2008. pages 92

[67] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff. A Hybrid Approach of OpenMP for Clusters. In *Proc. 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'12)*, pages 75–84, New Orleans, LA, USA, 2012. pages 6

[68] LAPACK and ScaLAPACK Survey. http://icl.cs.utk.edu/lapack-forum/survey/ [Last visited: November 2012]. pages 13, 16

[69] Linear Algebra PACKage (LAPACK). http://www.netlib.org/lapack/ [Last visited: November 2012]. pages 13, 78

[70] S. Liu, Y. Zhang, X. Sun, and R. Qiu. Performance Evaluation of Multithreaded Sparse Matrix-Vector Multiplication using OpenMP. In *Proc. 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'09)*, pages 659–665, Seoul, Korea, 2009. pages 92

[71] M. Luján, A. Usman, P. Hardie, L. Freeman, and J. R. Gurd. Storage Formats for Sparse Matrices in Java. In *Proc. 5th Intl. Conf. on Computational Science (ICCS'05)*, pages 364–371, Atlanta, GA, USA, 2005. pages 92

[72] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, volume 5759 of *Lecture Notes in Computer Science*, pages 174–184, Espoo, Finland, 2009. pages 1

[73] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, and G. Jin. A New Vision for Coarray Fortran. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, Ashburn, VA, USA, 2009. pages 8

[74] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115, Espoo, Finland, 2009. pages 2, 35, 45

[75] Message Passing Interface Forum. http://www.mpi-forum.org/ [Last visited: November 2012]. pages 6

[76] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a Parallel Language for Scientific Computation: Practice and Experience. In *Proc. 25th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'11)*, Anchorage, AK, USA, 2011. pages 9

[77] E. Musoll. Variable-Size Mosaics: A Process-Variation Aware Technique to Increase the Performance of Tile-Based, Massive Multi-Core Processors. *Computers and Electrical Engineering*, 37(3):1193–1211, 2011. pages 34

[78] NASA Advanced Computing Division. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/ [Last visited: November 2012]. pages 35, 49

[79] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick. Scaling Communication-Intensive Applications on BlueGene/P using One-Sided Communication and Overlap. In *Proc. 23rd IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'09)*, Rome, Italy, 2009. pages 1

[80] R. Nishtala, Y. Zheng, P. Hargrove, and K. Yelick. Tuning Collective Communication for Partitioned Global Address Space Programming Models. *Parallel Computing*, 37(9):576–591, 2011. pages 1

[81] R. W. Numrich. A Parallel Numerical Library for Co-array Fortran. In *Proc. Workshop on Language-Based Parallel Programming Models (WLPP'05)*, vol-

ume 3911 of *Lecture Notes in Computer Science*, pages 960–969, Poznan, Poland, 2005. pages 14

[82] R. W. Numrich. A Team Object for Coarray Fortran. In *Proc. 8th Intl. Conf. on Parallel Processing and Applied Mathematics (PPAM'09)*, volume 6068 of *Lecture Notes in Computer Science*, pages 68–73, Wroclaw, Poland, 2009. pages 8

[83] Parallel Basic Linear Algebra Subprograms (PBLAS) Library. http://www.netlib.org/scalapack/pblas_qref.html [Last visited: November 2012]. pages 13, 55, 78

[84] F. Patel and J. R. Gilbert. An Empirical Study of the Performance and Productivity of Two Parallel Programming Models. In *Proc. 22nd IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'08)*, Miami, FL, USA, 2008. pages 13

[85] Portable, Extensible Toolkit for Scientific Computation. http://www.mcs.anl.gov/petsc/ [Last visited: November 2012]. pages 97

[86] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Transactions on Mathematical Software*, 2012 (to appear). pages 13

[87] Project Fortress. http://projectfortress.java.net/ [Last visited: November 2012]. pages 9

[88] M. Püschel, J. M. F. Moura, J. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proc. of the IEEE*, 93(2):232–275, 2005. pages 34

[89] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP'09)*, pages 427–436, Weimar, Germany, 2009. pages 6

[90] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995. pages 36

[91] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based Global Load Balancing. In *Proc. 16th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'11)*, pages 201–212, San Antonio, TX, USA, 2011. pages 9

[92] W. N. Scherer, L. Adhianto, G. Jin, J. Mellor-Crummey, and C. Yang. Hiding Latency in Coarray Fortran 2.0. In *Proc. 4th Conf. on Partitioned Global Address Space Programming Models (PGAS'10)*, New York, NY, USA, 2010. pages 8

[93] R. Shahnaz and A. Usman. Blocked-Based Sparse Matrix-Vector Multiplication on Distributed Memory Parallel Computers. *The International Arab Journal of Information Technology*, 8(2):130–136, 2011. pages 92

[94] R. Shahnaz, A. Usman, and I. R. Chughtai. Implementation and Evaluation of Parallel Sparse Matrix-Vector Products on Distributed Memory Parallel Computers. In *Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006. pages 92

[95] H. Shan, F. Blagojevic, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A Programming Model Performance Study using the NAS Parallel Benchmarks. *Scientific Programming*, 18(3-4):153–167, 2010. pages 1

[96] H. Shan, N. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. In *Proc. 2nd Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS'11)*, pages 13–14, Seattle, WA, USA, 2011. pages 1

[97] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. A. Padua. Performance Portability with the Chapel Language. In *Proc. 26th IEEE*

Intl. Parallel and Distributed Processing Symp. (IPDPS'12), Shanghai, China, 2012. pages 8

[98] S. Sistare, R. Vandevaart, and E. Loh. Optimization of MPI Collectives on Clusters of Large-Scale SMPs. In *Proc. 11th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'99)*, Portland, OR, USA, 1999. pages 2, 34

[99] Sparse Basic Linear Algebra Subprograms (SparseBLAS) Library. http://math.nist.gov/spblas [Last visited: November 2012]. pages 12, 92, 98

[100] STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/ [Last visited: November 2012]. pages 42

[101] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC Programmability using Classroom Studies. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, Ashburn, VA, USA, 2009. pages 13

[102] The Chapel Parallel Programming Language. http://chapel.cray.com/ [Last visited: November 2012]. pages 8

[103] The IBM Engineering Scientific Subroutine Library (ESSL) and Parallel ESSL. http://www-03.ibm.com/systems/p/software/essl/index.html [Last visited: November 2012]. pages 13

[104] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/ [Last visited: November 2012]. pages 6

[105] The ScaLAPACK Project. http://netlib2.cs.utk.edu/scalapack/index.html [Last visited: November 2012]. pages 13, 79

[106] The Servet Benchmark Suite Homepage. http://servet.des.udc.es/ [Last visited: November 2012]. pages 159

[107] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/ [Last visited: November 2012]. pages 101

[108] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast Collective Operations using Shared and Remote Memory Access Protocols on Clusters. In *Proc. 17th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'03)*, Nice, France, 2003. pages 2, 34

[109] Titanium Project. http://titanium.cs.berkeley.edu/ [Last visited: November 2012]. pages 1, 8

[110] TOP500 List - June 2012. http://www.top500.org/list/2012/06/100 [Last visited: November 2012]. pages 54

[111] N. Travinin and J. Kepner. pMatlab Parallel Matlab Library. *Intl. Journal of High Performance Computing Applications*, 21(3):336–359, 2007. pages 14

[112] UPC Consortium. UPC Language Specifications, v1.2. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf [Last visited: November 2012]. pages 1, 17, 30

[113] UPCBLAS: A Numerical Library for Unified Parallel C. http://upcblas.des.udc.es/ [Last visited: November 2012]. pages 125

[114] A. Usman, M. Luján, L. Freeman, and J. R. Gurd. Performance Evaluation of Storage Formats for Sparse Matrices in Fortran. In *Proc. 8th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'06)*, pages 160–169, Munich, Germany, 2006. pages 92

[115] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Automatically Tuned Collective Communications. In *Proc. 12th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'00)*, Dallas, TX, USA, 2000. pages 34

[116] F. Vázquez, J. J. Fernández, and E. M. Garzón. Automatic Tuning of the Sparse Matrix Vector Product on GPUs Based on the ELLR-T Approach. *Parallel Computing*, 38(8):408–420, 2012. pages 92

[117] L. Wang, S. Merchant, and T. El-Ghazawi. Exploiting Hierarchical Parallelism using UPC. In *Proc. 25th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'11)*, Anchorage, AK, USA, 2011. pages 2

[118] T. Wen and P. Colella. Adaptive Mesh Refinement in Titanium. In *Proc. 19th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'05)*, Denver, CO, USA, 2005. pages 8

[119] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen. An Adaptive Mesh Refinement Benchmark for Modern Parallel Programming Languages. In *Proc. 19th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'07)*, Reno, NV, USA, 2007. pages 8

[120] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001. pages 34

[121] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Proc. 19th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'07)*, Reno, NV, USA, 2007. pages 91, 100

[122] X10: Performance and Productivity at Scale. http://x10-lang.org/ [Last visited: November 2012]. pages 8

[123] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and Performance using Partitioned Global Address Space Languages. In *Proc. 5th Intl. Workshop on Parallel Symbolic Computation (PASCO'07)*, pages 24–32, London, Canada, 2007. pages 6

[124] K. Yotov, X. Li, G. Ren, M. J. Garzarán, D. A. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High Performance BLAS? *Proc. of the IEEE*, 93(2):358–386, 2005. pages 34

[125] K. Yotov, K. Pingali, and P. Stodghill. Automatic Measurement of Memory Hierarchy Parameters. In *Proc. ACM Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*, pages 181–192, Banff, Canada, 2005. pages 35, 36

[126] K. Yotov, K. Pingali, and P. Stodghill. X-Ray: A Tool for Automatic Measurement of Hardware Parameters. In *Proc. 2nd Intl. Conf. on the Quantitative Evaluation of Systems (QEST'05)*, pages 168–177, Torino, Italy, 2005. pages 35

[127] J. Zhang, J. Zhai, W. Chen, and W. Zheng. Process Mapping for Collective Communications. In *Proc. 15th Intl. European Conf. on Parallel and Distributed Computing (Euro-Par'09)*, volume 5704 of *Lecture Notes in Computer Science*, pages 81–92, Delft, The Netherlands, 2009. pages 2, 34

[128] Y. Zheng. Optimizing UPC Programs for Multi-Core Systems. *Scientific Programming*, 18(3-4):183–191, 2011. pages 30

Appendix A

UPCBLAS Interface

This appendix shows the interface of all the routines included in UPCBLAS. It begins with a description of the enumerated values specific of the library and it continues providing the syntax of the routines, an explanation of the meaning of their parameters and some hints to obtain correct results and good performance. This appendix is part of the UPCBLAS Reference Manual that is available in [113], together with the last version of the library.

A.1. Enumerated Values

The UPCBLAS routines use enumerated values to specify some characteristics of the input matrices. Their declaration is available in the file *include/types.h*.

- `UPC_PBLAS_DIMMDIST`: It indicates if the matrix is distributed by rows or columns.
    - `upc_pblas_rowDist`
    - `upc_pblas_colDist`
- `UPC_PBLAS_TRANSPOSE`: It indicates if the matrix is transposed.
    - `upc_pblas_noTrans`
    - `upc_pblas_trans`

  - `upc_pblas_conjTrans`

- `UPC_PBLAS_UPLO`: It indicates if the matrix is upper or lower triangular.
  - `upc_pblas_upper`
  - `upc_pblas_lower`
- `UPC_PBLAS_DIAG`: It indicates if all the elements in the main diagonal of the triangular matrix are equal to 1 or not.
  - `upc_pblas_nonUnit`
  - `upc_pblas_unit`
- `UPC_PBLAS_SIDE`: In the BLAS3 triangular solver it indicates if the triangular matrix is on the left or on the right side of the equation.
  - `upc_pblas_left`
  - `upc_pblas_right`

A.2. BLAS1 Routines

These routines perform vector-vector operations. Their headers are available in the file *include/pblas1.h*.

A.2.1. upc_blas_Tcopy

Function to copy vector `x` to vector `y`.

SYNTAX:

- `int upc_blas_scopy(int block_size, int size, shared void *x, shared void *y)`
- `int upc_blas_dcopy(int block_size, int size, shared void *x, shared void *y)`

- `int upc_blas_ccopy(int block_size, int size, shared void *x, shared void *y)`
- `int upc_blas_zcopy(int block_size, int size, shared void *x, shared void *y)`

PARAMETERS:

- `IN block_size`: Storage block size for vectors `x` and `y`.
- `IN size`: Vectors length.
- `IN x`: Pointer to the position of the shared array where the source vector `x` is stored.
- `OUT y`: Pointer to the position of the shared array where the destination vector `y` is stored.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

RESTRICTIONS:

- This function treats pointers `x` and `y` as if they had type `shared [block_size] type[size]`.
- The address of the first element of `x` and `y` must have phase 0.
- The first element of `x` and `y` must be in parts of the shared memory with affinity to the same thread.
- If `x` or `y` overlap, the behavior is undefined.

A.2.2. upc_blas_Tswap

Function to swap the elements of two vectors.

SYNTAX:

- `int upc_blas_sswap(int block_size, int size, shared void *x, shared void *y)`
- `int upc_blas_dswap(int block_size, int size, shared void *x, shared void *y)`
- `int upc_blas_cswap(int block_size, int size, shared void *x, shared void *y)`
- `int upc_blas_zswap(int block_size, int size, shared void *x, shared void *y)`

PARAMETERS:

- `IN block_size`: Storage block size for the vectors to swap (`x` and `y`).
- `IN size`: Vectors length.
- `IN/OUT x,y`: Pointers to the positions of the shared arrays where vectors `x` and `y` are stored.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

RESTRICTIONS:

- This function treats pointers `x` and `y` as if they had type `shared [block_size] type[size]`.

- The address of the first element of `x` and `y` must have phase 0.
- The first element of `x` and `y` must be in parts of the shared memory with affinity to the same thread.
- If `x` or `y` overlap, the behavior is undefined.

A.2.3. upc_blas_Tscal

Function to scale a vector by a scalar.

<u>SYNTAX:</u>

- `int upc_blas_sscal(int block_size, int size, float alpha, shared void *x)`
- `int upc_blas_dscal(int block_size, int size, double alpha, shared void *x)`
- `int upc_blas_cscal(int block_size, int size, void *alpha, shared void *x)`
- `int upc_blas_zscal(int block_size, int size, void *alpha, shared void *x)`
- `int upc_blas_csscal(int block_size, int size, float alpha, shared void *x)`
- `int upc_blas_zdscal(int block_size, int size, double alpha, shared void *x)`

<u>PARAMETERS:</u>

- `IN block_size`: Storage block size for the vector.
- `IN size`: Vector length.

- `IN alpha`: Scale factor.
- `IN/OUT x`: Pointer to the position of the shared array where vector `x` is stored.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

RESTRICTIONS:

- This function treats pointer `x` as if it had type `shared [block_size] type[size]`.
- The address of the first element of `x` must have phase 0.

A.2.4. upc_blas_Taxpy

Function to update a vector by adding to it another vector scaled by a factor. These vectors are stored in shared arrays. The equivalent function is: $y = \alpha * x + y$.

SYNTAX:

- `int upc_blas_saxpy(int block_size, int size, float alpha, shared void *x, shared void *y)`
- `int upc_blas_daxpy(int block_size, int size, double alpha, shared void *x, shared void *y)`
- `int upc_blas_caxpy(int block_size, int size, void * alpha, shared void *x, shared void *y)`
- `int upc_blas_zaxpy(int block_size, int size, void * alpha, shared void *x, shared void *y)`

PARAMETERS:

- `IN block_size`: Storage block size for the vectors.
- `IN size`: Vectors length.
- `IN alpha`: Scale factor.
- `IN x`: Pointer to the position of the shared array where the source vector `x` is stored.
- `IN/OUT y`: Pointer to the position of the shared array where vector `y` is stored.
- returns:
    - 0 if everything is ok.
    - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
    - +1 if an internal memory error occurs.

RESTRICTIONS:

- This function treats pointers `x` and `y` as if they had type `shared [block_size] type[size]`.
- The address of the first element of `x` and `y` must have phase 0.
- The first element of `x` and `y` must be in parts of the shared memory with affinity to the same thread.
- If `x` or `y` overlap, the behavior is undefined.

A.2.5. upc_blas_Tsdot

Function to perform the dot product between two vectors stored in shared arrays.

SYNTAX:

- `int upc_blas_sdot(int block_size, int size, shared void *x, shared void *y, shared float *dst)`
- `int upc_blas_ddot(int block_size, int size, shared void *x, shared void *y, shared double *dst)`
- `int upc_blas_cdotc(int block_size, int size, shared void *x, shared void *y, shared void *_dst)`
- `int upc_blas_zdotc(int block_size, int size, shared void *x, shared void *y, shared void *_dst)`
- `int upc_blas_cdotu(int block_size, int size, shared void *x, shared void *y, shared void *_dst)`
- `int upc_blas_zdotu(int block_size, int size, shared void *x, shared void *y, shared void *_dst)`

PARAMETERS:

- `IN block_size`: Storage block size for the vectors.
- `IN size`: Vectors length.
- `IN x,y`: Pointers to the positions of the shared arrays where the source vectors `x` and `y` are stored.
- `OUT dst/_dst`: Pointer to the position of shared memory where the dot product result is stored. This pointer must have memory allocated for one element.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

RESTRICTIONS:

- This function treats pointers `x` and `y` as if they had type `shared [block_size] type[size]`.
- The address of the first element of `x` and `y` must have phase 0.
- The first element of `x` and `y` must be in parts of the shared memory with affinity to the same thread.
- If `x` or `y` overlap, the behavior is undefined.

A.2.6. upc_blas_Tnrm2

Function to perform the euclidean norm of a vector stored in a shared array.

SYNTAX:

- `int upc_blas_snrm2(int block_size, int size, shared void *x, shared float *dst)`
- `int upc_blas_dnrm2(int block_size, int size, shared void *x, shared double *dst)`
- `int upc_blas_scnrm2(int block_size, int size, shared void * x, shared float *dst)`
- `int upc_blas_dznrm2(int block_size, int size, shared void * x, shared double *dst)`

PARAMETERS:

- `IN block_size`: Storage block size for vector `x`.
- `IN size`: Vector length.
- `IN x`: Pointer to the position of the shared array where the source vector `x` is stored.

- `OUT dst`: Pointer to the position of shared memory where the norm result is stored. This pointer must have memory allocated for one element.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

RESTRICTIONS:

- This function treats pointer `x` as if it had type `shared [block_size] type[size]`.
- The address of the first element of `x` must have phase 0.

A.2.7. upc_blas_Tasum

Function to perform the sum of the absolute values of all the elements of a vector.

SYNTAX:

- `int upc_blas_sasum(int block_size, int size, shared void * x, shared float * dst)`
- `int upc_blas_dasum(int block_size, int size, shared void * x, shared double * dst)`
- `int upc_blas_scasum(int block_size, int size, shared void * x, shared float * dst)`
- `int upc_blas_dzasum(int block_size, int size, shared void * x, shared double * dst)`

PARAMETERS:

- `IN block_size`: Storage block size for vector `x`.
- `IN size`: Vector length.
- `IN x`: Pointer to the position of the shared array where the source vector `x` is stored.
- `OUT dst`: Pointer to the position of shared memory where the sum result is stored. This pointer must have memory allocated for one element.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

<u>RESTRICTIONS:</u>

- This function treats pointer `x` as if it had type `shared [block_size] type[size]`.
- The address of the first element of `x` must have phase 0.

A.3. BLAS2 Routines

These routines perform matrix-vector operations. Their headers are available in the file *include/pblas2.h*.

A.3.1. upc_blas_Tgemv

Function to perform the matrix-vector product: $y = \alpha * A * x + \beta * y$ (with transpose variants of matrix `A`).

<u>SYNTAX:</u>

- `int upc_blas_sgemv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transpose, int m, int n, float alpha, shared void *A, int lda, shared void *x, float beta, shared void *y)`

- `int upc_blas_dgemv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transpose, int m, int n, double alpha, shared void *A, int lda, shared void *x, double beta, shared void *y)`

- `int upc_blas_cgemv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transpose, int m, int n, void *alpha, shared void *A, int lda, shared void *x, void beta, shared void *y)`

- `int upc_blas_zgemv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transpose, int m, int n, void *alpha, shared void *A, int lda, shared void *x, void beta, shared void *y)`

PARAMETERS:

- `IN dimmDist`: Enumerated value representing the matrix dimension that will be distributed among threads. Each thread must have one or more complete rows or columns depending on this parameter. If `upc_pblas_rowDist` is selected, each thread has complete rows. Columns are distributed entirely in the `upc_pblas_colDist` case.

- `IN block_size`: Number of rows or columns (depending on `dimmDist`) of the matrix that are consecutively distributed among all threads. For example, in the `upc_pblas_rowDist` case, the first `block_size` rows will correspond to thread 0, the second `block_size` ones to thread 1, etc. Remember that blocks are built using whole rows/columns and not individual elements.

- `IN sec_block_size`: Depending on the selected `dimmDist`, the block size of one of the source vectors is automatically determined (see RESTRICTIONS below) but the distribution of the other vector can vary. This parameter is used to indicate the block size of the vector not automatically defined.

- `IN transpose`: Enumerated value to indicate if the matrix is transposed.
- `IN m`: Number of rows of matrix `A`.
- `IN n`: Number of columns of matrix `A`.
- `IN alpha`: Scale factor for matrix `A`.
- `IN A`: Pointer to the position of the shared array where the source matrix `A` is stored.
- `IN lda`: It specifies the first dimension of `A` as declared in the calling program (used to work with submatrices). It must be at least `n`.
- `IN x`: Pointer to the position of the shared array where the source vector `x` is stored.
- `IN beta`: Scale factor for vector `y`.
- `IN/OUT y`: Pointer to the position of the shared array where vector `y` is stored.
- returns:
  - 0 if everything is ok.
  - <0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

RESTRICTIONS:

The block size specifications are different depending on the `dimmDist` selection:

1 `upc_pblas_rowDist`: All the elements in a row must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- `A:shared [block_size*lda] type [m*lda]`
- `x:shared [sec_block_size] type [n]`
- `y:shared [block_size] type [m]`

In transpose case:

- `A:shared [block_size] type [m*lda]`
- `x:shared [sec_block_size] type [m]`
- `y:shared [block_size] type [n]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- The address of the first element of `y` must have phase 0.
- The address of the first element of `x` must have phase 0.
- The first element of `A` and `y` must be in parts of the shared memory with affinity to the same thread.
- If `A` non-transpose:
  - The first element of `A` must be in the first row of a block in the shared memory space.
- If `A` transpose:
  - The address of the first element of `A` must have phase 0.
  - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $lda\%(block\_size * THREADS) == 0$.

2 `upc_pblas_colDist`: All the elements in a column must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- `A:shared [block_size] type [m*lda]`
- `x:shared [block_size] type [n]`
- `y:shared [sec_block_size] type [m]`

In transpose case:

- `A:shared [block_size*lda] type [m*lda]`
- `x:shared [block_size] type [m]`
- `y:shared [sec_block_size] type [n]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- The address of the first element of `y` must have phase 0.
- The address of the first element of `x` must have phase 0.
- The first element of `A` and `x` must be in parts of the shared memory with affinity to the same thread.
- If `A` non-transpose:
  - The address of the first element of `A` must have phase 0.
  - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $lda\%(block\_size * THREADS) == 0$.
- If `A` transpose:
  - The first element of `A` must be in the first row of a block in the shared memory space.

If any array overlaps, the behavior is undefined.

ADVICE:

The `upc_pblas_rowDist` case is much more efficient than the `upc_pblas_colDist` case.

A.3.2. upc_blas_Tger

Function to perform the outer product of two vectors: $A = \alpha * x * y' + A$.

SYNTAX:

- `int upc_blas_sger(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, int m, int n, float alpha, shared void *x, shared void *y, shared void *A, int lda)`
- `int upc_blas_dger(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, int m, int n, double alpha, shared void *x, shared void *y, shared void *A, int lda)`
- `int upc_blas_cgerc(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, int m, int n, void *alpha, shared void *x, shared void *y, shared void *A, int lda)`
- `int upc_blas_zgerc(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, int m, int n, void *alpha, shared void *x, shared void *y, shared void *A, int lda)`
- `int upc_blas_cgeru(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, int m, int n, void *alpha, shared void *x, shared void *y, shared void *A, int lda)`
- `int upc_blas_zgeru(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, int m, int n, void *alpha, shared void *x, shared void *y, shared void *A, int lda)`

PARAMETERS:

- `IN dimmDist`: Enumerated value representing the matrix dimension that will be distributed among threads. Each thread must have one or more complete rows or columns depending on this parameter. If `upc_pblas_rowDist` is selected, each thread has complete rows. Columns are distributed entirely in the `upc_pblas_colDist` case.
- `IN block_size`: Number of rows or columns (depending on `dimmDist`) of the matrix that are consecutively distributed among all threads. For example, in the `upc_pblas_rowDist` case, the first `block_size` rows will correspond to

thread 0, the second `block_size` ones to thread 1, etc. Remember that blocks are built using whole rows/columns and not individual elements.

- `IN sec_block_size`: Depending on the selected `dimmDist`, the block size of one of the source vectors is automatically determined (see RESTRICTIONS below) but the distribution of the other vector can vary. This parameter is used to indicate the block size of the vector not automatically defined.
- `IN m`: Number of rows of matrix `A`.
- `IN n`: Number of columns of matrix `A`.
- `IN alpha`: Scale factor.
- `IN x`: Pointer to the position of the shared array where the source vector `x` is stored.
- `IN y`: Pointer to the position of the shared array where the source vector `y` is stored.
- `IN/OUT A`: Pointer to the position of the shared array where matrix `A` is stored.
- `IN lda`: It specifies the first dimension of `A` as declared in the calling program (used to work with submatrices). It must be at least `n`.
- returns:
  - 0 if everything is ok.
  - $<$ 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

<u>RESTRICTIONS:</u>

The block size specifications are different depending on the `dimmDist` selection:

1 `upc_pblas_rowDist`: All the elements in a row must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- `A:shared [block_size*lda] type [m*lda]`
- `x:shared [sec_block_size] type [m]`
- `y:shared [block_size] type [n]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- The address of the first element of `y` must have phase 0.
- The address of the first element of `x` must have phase 0.
- The first element of `A` and `x` must be in parts of the shared memory with affinity to the same thread.
- The first element of `A` must be in the first row of a block in the shared memory space.

2 `upc_pblas_colDist`: All the elements in a column must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- `A:shared [block_size] type [m*lda]`
- `x:shared [block_size] type [m]`
- `y:shared [sec_block_size] type [n]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- The address of the first element of `y` must have phase 0.
- The address of the first element of `x` must have phase 0.
- The first element of `A` and `y` must be in parts of the shared memory with affinity to the same thread.
- The address of the first element of `A` must have phase 0.
- The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $lda\%(block\_size * THREADS) == 0$.

If any array overlaps, the behavior is undefined.

A.3.3. upc_blas_Ttrsv

Function to solve the triangular system of equations $x = T^{-1} * x$ (with transpose variants of matrix `T`).

<u>SYNTAX:</u>

- `int upc_blas_strsv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int n, shared void *T, int ldt, shared void *x)`
- `int upc_blas_dtrsv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int n, shared void *T, int ldt, shared void *x)`
- `int upc_blas_ctrsv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int n, shared void *T, int ldt, shared void *x)`
- `int upc_blas_ztrsv(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int n, shared void *T, int ldt, shared void *x)`

<u>PARAMETERS:</u>

- `IN dimmDist`: Enumerated value representing the matrix dimension that will be distributed among threads. Each thread must have one or more complete rows or columns depending on this parameter. If `upc_pblas_rowDist` is selected, each thread has complete rows. Columns are distributed entirely in the `upc_pblas_colDist` case.
- `IN block_size`: Number of rows or columns (depending on `dimmDist`) of the matrix that are consecutively distributed among all threads. For example, in the `upc_pblas_rowDist` case, the first `block_size` rows will correspond to thread 0, the second `block_size` ones to thread 1, etc. Remember that blocks are built using whole rows/columns and not individual elements.

- `IN uplo`: Enumerated value to indicate if matrix `T` is upper or lower triangular.
- `IN transpose`: Enumerated value to indicate if matrix `T` is transposed.
- `IN diag`: Enumerated value to indicate if all the diagonal values of `T` are equal to 1 or not.
- `IN n`: Number of rows and columns of matrix `T`.
- `IN T`: Pointer to the position of the shared array where the source triangular matrix `T` is stored.
- `IN ldt`: It specifies the first dimension of `T` as declared in the calling program (used to work with submatrices). It must be at least `n`.
- `IN/OUT x`: Pointer to the position of the shared array where vector `x` is stored.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

RESTRICTIONS:

The block size specifications are different depending on the `dimmDist` selection:

1 `upc_pblas_rowDist`: All the elements in a row must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- `T:shared [block_size*ldt] type [n*ldt]`
- `T:shared [block_size] type [n*ldt]` in transpose case
- `x:shared [block_size] type [n]`

In addition, there are some limitations about the `block_size` parameter:

- The address of the first element of `x` must have phase 0.
- The first element of `T` and `x` must be in parts of the shared memory with affinity to the same thread.
- If `T` non-transpose:
  - The first element of `T` must be in the first row of a block in the shared memory space.
- If `T` transpose:
  - The address of the first element of `T` must have phase 0.
  - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $ldt\%(block\_size * THREADS) == 0$.

2 `upc_pblas_colDist`: All the elements in a column must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- `T:shared [block_size] type [n*ldt]`
- `T:shared [block_size*ldt] type [n*ldt]` in transpose case
- `x:shared [block_size] type [n]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- The address of the first element of `x` must have phase 0.
- The first element of `T` and `x` must be in parts of the shared memory with affinity to the same thread.
- If `T` non-transpose:
  - The address of the first element of `T` must have phase 0.
  - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $ldt\%(block\_size * THREADS) == 0$.
- If `T` transpose:

- The first element of `T` must be in the first row of a block in the shared memory space.

If any array overlaps, the behavior is undefined.

ADVICE:

No check for singularity or near singularity of matrix `T` is included in this function. It must be implemented before the function call if necessary.

A.4. BLAS3 Routines

These routines perform matrix-matrix operations. Their headers are available in the file *include/pblas3.h*.

A.4.1. upc_blas_Tgemm

Function to perform the matrix-matrix product: $C = \alpha * A * B + \beta * C$ (with transpose variants of matrices `A` and `B`).

SYNTAX:

- `int upc_blas_sgemm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transposeA, UPC_PBLAS_TRANSPOSE transposeB, int m, int n, int k, float alpha, shared void *A, int lda, shared void *B, int ldb, float beta, shared void *C, int ldc)`
- `int upc_blas_dgemm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transposeA, UPC_PBLAS_TRANSPOSE transposeB, int m, int n, int k, double alpha, shared void *A, int lda, shared void *B, int ldb, double beta, shared void *C, int ldc)`

- `int upc_blas_cgemm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transposeA, UPC_PBLAS_TRANSPOSE transposeB, int m, int n, int k, void alpha, shared void *A, int lda, shared void *B, int ldb, void beta, shared void *C, int ldc)`

- `int upc_blas_zgemm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_TRANSPOSE transposeA, UPC_PBLAS_TRANSPOSE transposeB, int m, int n, int k, void alpha, shared void *A, int lda, shared void *B, int ldb, void beta, shared void *C, int ldc)`

PARAMETERS:

- `IN dimmDist`: Enumerated value representing the dimension of matrix `C` that will be distributed among threads. Each thread must have one or more complete rows or columns depending on this parameter. If `upc_pblas_rowDist` is selected, each thread has complete rows. Columns are distributed entirely in `upc_pblas_colDist` case.

- `IN block_size`: Number of rows or columns (depending on `dimmDist`) of matrix `C` that are consecutively distributed among all threads. For example, in the `upc_pblas_rowDist` case, the first `block_size` rows will correspond to thread 0, the second `block_size` ones to thread 1, etc. Remember that blocks are built using whole rows/columns and not individual elements.

- `IN sec_block_size`: Depending on the selected `dimmDist`, the block size of one of the source matrices (`A` or `B`) is automatically determined (see RESTRICTIONS below) but the distribution of the other matrix can vary. This parameter is used to indicate the block size of the matrix not automatically defined.

- `IN transposeA`: Enumerated value to indicate if matrix `A` is transposed.

- `IN transposeB`: Enumerated value to indicate if matrix `B` is transposed.

- `IN m`: Number of rows of matrices `A` and `C`.

- `IN n`: Number of columns of matrices `B` and `C`.

- `IN k`: Number of columns of matrix `A` and rows of matrix `B`.
- `IN alpha`: Scale factor for matrix `A`.
- `IN A`: Pointer to the position of the shared array where the source matrix `A` is stored.
- `IN lda`: It specifies the first dimension of `A` as declared in the calling program (used to work with submatrices). It must be at least `k` in the non-transpose case and `m` in the transpose case.
- `IN B`: Pointer to the position of the shared array where the source matrix `B` is stored.
- `IN ldb`: It specifies the first dimension of `B` as declared in the calling program (used to work with submatrices). It must be at least `n` in the non-transpose case and `k` in the transpose case.
- `IN beta`: Scale factor for matrix `C`.
- `IN/OUT C`: Pointer to the position of the shared array where matrix `C` is stored.
- `IN ldc`: It specifies the first dimension of `C` as declared in the calling program (used to work with submatrices). It must be at least `n`.
- returns:
  - 0 if everything is ok.
  - <0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

<u>RESTRICTIONS:</u>

The block size specifications are different depending on the `dimmDist` selection:

1 `upc_pblas_rowDist`: All the elements in a row of matrix `C` must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- A:shared [block_size*lda] type [m*lda]
- A:shared [block_size] type [k*lda] in transpose case
- B:shared [sec_block_size] type [k*ldb]
- B:shared [sec_block_size] type [n*ldb] in transpose case
- C:shared [block_size*ldc] type [m*ldc]

In addition, there are some limitations about the block_size and sec_block_size parameters:

- ldb and sec_block_size must be multiples of each other.
- The first element of B must be in the first row of a block in the shared memory space.
- The first element of C must be in the first row of a block in the shared memory space.
- The first element of A and C must be in parts of the shared memory with affinity to the same thread.
- If A non-transpose:
  - The first element of A must be in the first row of a block in the shared memory space.
- If A transpose:
  - The first element of A must be in the first column of a block in the shared memory space.
  - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $lda\%(block\_size * THREADS) == 0$.

2 upc_pblas_colDist: All the elements in a column of matrix C must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- A:shared [sec_block_size] type [m*lda]
- A:shared [sec_block_size] type [k*lda] in transpose case

- `B:shared [block_size] type [k*ldb]`
- `B:shared [block_size*ldb] type [n*ldb]` in transpose case
- `C:shared [block_size] type [m*ldc]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- `lda` and `sec_block_size` must be multiples of each other.
- The first element of `A` must be in the first row of a block in the shared memory space.
- The first element of `C` must be in the first column of a block in the shared memory space.
- The first element of `B` and `C` must be in parts of the shared memory with affinity to the same thread.
- If `B` non-transpose:
  - The first element of `B` must be in the first column of a block in the shared memory space.
  - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $ldb\%(block\_size * THREADS) == 0$.
- If `B` transpose:
  - The first element of `B` must be in the first row of a block in the shared memory space.

If any array overlaps, the behavior is undefined.

A.4.2. upc_blas_Ttrsm

Function to solve the triangular systems of equations $X = T^{-1} * \alpha * X$ or $X = \alpha * X * T^{-1}$ (with transpose variants of matrix `T`).

SYNTAX:

- `int upc_blas_strsm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_SIDE side, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int m, int n, float alpha, shared void *T, int ldt, shared void *X, int ldx);`
- `int upc_blas_dtrsm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_SIDE side, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int m, int n, double alpha, shared void *T, int ldt, shared void *X, int ldx);`
- `int upc_blas_ctrsm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_SIDE side, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int m, int n, void *alpha, shared void *T, int ldt, shared void *X, int ldx);`
- `int upc_blas_ztrsm(UPC_PBLAS_DIMMDIST dimmDist, int block_size, int sec_block_size, UPC_PBLAS_SIDE side, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, UPC_PBLAS_DIAG diag, int m, int n, void *alpha, shared void *T, int ldt, shared void *X, int ldx);`

PARAMETERS:

- `IN dimmDist`: Enumerated value representing the dimension of matrix `X` that will be distributed among threads. Each thread must have one or more complete rows or columns depending on this parameter. If `upc_pblas_rowDist` is selected, each thread has complete rows. Columns are distributed entirely in the `upc_pblas_colDist` case.
- `IN block_size`: Number of rows or columns (depending on `dimmDist`) of matrix `X` that are consecutively distributed among all threads. For example, in the `upc_pblas_rowDist` case, the first `block_size` rows will correspond to thread 0, the second `block_size` ones to thread 1, etc. Remember that blocks are built using whole rows/columns and not individual elements.
- `IN sec_block_size`: Number of elements consecutively distributed among all threads in matrix `T` in the options: `upc_pblas_colDist & upc_pblas_left` or `upc_pblas_rowDist & upc_pblas_right`.

- `IN side`: Enumerated value to indicate if matrix `T` is on the left or right side of the equation.
- `IN uplo`: Enumerated value to indicate if matrix `T` is upper or lower triangular.
- `IN transpose`: Enumerated value to indicate if matrix `T` is transposed.
- `IN diag`: Enumerated value to indicate if all the diagonal values of `T` are equal to 1 or not.
- `IN m`: Number of rows of matrix `X`. Dimension of matrix `T` if `upc_pblas_left`.
- `IN n`: Number of columns of matrix `X`. Dimension of matrix `T` if `upc_pblas_right`.
- `IN alpha`: Scale factor for matrix `T`.
- `IN T`: Pointer to the position of the shared array where the source triangular matrix `T` is stored.
- `IN ldt`: It specifies the first dimension of `T` as declared in the calling program (used to work with submatrices). It must be at least `m` in the `upc_pblas_left` case and `n` in the `upc_pblas_right` case.
- `IN/OUT X`: Pointer to the position of the shared array where matrix `X` is stored.
- `IN ldx`: It specifies the first dimension of `X` as declared in the calling program (used to work with submatrices). It must be at least `n`.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

RESTRICTIONS:

The block size specifications are different depending on the `dimmDist` and `side` selection:

1 `upc_pblas_rowDist & upc_pblas_left`: All the elements in a row of matrix `X` must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- `T:shared [block_size*ldt] type [m*ldt]`
- `T:shared [block_size] type [m*ldt]` in transpose case
- `X:shared [block_size*ldx] type [m*ldx]`

In addition, there are some limitations about the `block_size` parameter:

- The first element of `X` must be in the first row of a block in the shared memory space.
- The first element of `T` and `X` must be in parts of the shared memory with affinity to the same thread.
- If `T` non-transpose:
  - The first element of `T` must be in the first row of a block in the shared memory space.
- If `T` transpose:
  - The first element of `T` must be in the first column of a block in the shared memory space.

2 `upc_pblas_colDist & upc_pblas_left`: All the elements in a column of matrix `X` must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- `T:shared [sec_block_size] type [m*ldt]`
- `X:shared [block_size] type [m*ldx]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- `ldt` and `sec_block_size` must be multiples of each other.

- The first element of `T` must be in the first row of a block in the shared memory space.
- The first element of `X` must be in the first column of a block in the shared memory space.

3 `upc_pblas_rowDist & upc_pblas_right`: All the elements in a row of matrix `X` must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- `T:shared [sec_block_size] type [n*ldt]`
- `X:shared [block_size*ldx] type [m*ldx]`

In addition, there are some limitations about the `block_size` and `sec_block_size` parameters:

- `ldt` and `sec_block_size` must be multiples of each other.
- The first element of `T` must be in the first row of a block in the shared memory space.
- The first element of `X` must be in the first row of a block in the shared memory space.

4 `upc_pblas_colDist & upc_pblas_right`: All the elements in a column of matrix `X` must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- `T:shared [block_size] type [n*ldt]`
- `T:shared [block_size*ldt] type [n*ldt]` in transpose case
- `X:shared [block_size] type [m*ldx]`

In addition, there are some limitations about the `block_size` parameter:

- The first element of `X` must be in the first column of a block in the shared memory space.

- The first element of `T` and `X` must be in parts of the shared memory with affinity to the same thread.
- If `T` non-transpose:
  - The first element of `T` must be in the first column of a block in the shared memory space.
- If `T` transpose:
  - The first element of `T` must be in the first row of a block in the shared memory space.

If any array overlaps, the behavior is undefined.

No check for singularity or near singularity of matrix `T` is included in this function. It must be implemented before the function call if necessary.

A.4.3. upc_blas_Tsyrk

Function to perform the product of a symmetric matrix by its transpose: $C = \alpha * A * A' + \beta * C$ (with transpose variants of matrix `A`).

SYNTAX:

- `int upc_blas_ssyrk(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, int n, int k, float alpha, shared void *A, int lda, float beta, shared void *C, int ldc);`
- `int upc_blas_dsyrk(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, int n, int k, double alpha, shared void *A, int lda, double beta, shared void *C, int ldc);`
- `int upc_blas_csyrk(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, int n, int k,`

```
void *alpha, shared void *A, int lda, void *beta, shared
void *C, int ldc);
```

- `int upc_blas_zsyrk(UPC_PBLAS_DIMMDIST dimmDist, int block_size, UPC_PBLAS_UPLO uplo, UPC_PBLAS_TRANSPOSE transpose, int n, int k, void *alpha, shared void *A, int lda, void *beta, shared void *C, int ldc);`

PARAMETERS:

- `IN dimmDist`: Enumerated value representing the dimension of matrix `C` that will be distributed among threads. Each thread must have one or more complete rows or columns depending on this parameter. If `upc_pblas_rowDist` is selected, each thread has complete rows. Columns are distributed entirely in `upc_pblas_colDist` case.
- `IN block_size`: Number of rows or columns (depending on `dimmDist`) of matrices `A` and `C` that are consecutively distributed among all threads. For example, in the `upc_pblas_rowDist` case, the first `block_size` rows will correspond to thread 0, the second `block_size` ones to thread 1, etc. Remember that blocks are built using whole rows/columns and not individual elements.
- `IN uplo`: Enumerated value to indicate if the elements of `A` are stored in the upper or lower half of the matrix.
- `IN transpose`: Enumerated value to indicate if matrix `A` is transposed.
- `IN n`: Number of rows of matrix `C` and rows and columns of matrix `A`.
- `IN k`: Number of columns of matrix `C`.
- `IN alpha`: Scale factor for matrix `A`.
- `IN A`: Pointer to the position of the shared array where the symmetric matrix `A` is stored.
- `IN lda`: It specifies the first dimension of `A` as declared in the calling program (used to work with submatrices). It must be at least `n`.

- `IN beta`: Scale factor for matrix `C`.
- `IN/OUT C`: Pointer to the position of the shared array where matrix `C` is stored.
- `IN ldc`: It specifies the first dimension of `C` as declared in the calling program (used to work with submatrices). It must be at least `k`.
- returns:
  - 0 if everything is ok.
  - $<$ 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.
  - +1 if an internal memory error occurs.

RESTRICTIONS:

The block size specifications are different depending on the `dimmDist` selection:

1 `upc_pblas_rowDist`: All the elements in a row of matrix `C` must have affinity to the same thread but the number of consecutive rows in each thread can change. In this case, this function treats array pointers as if they had type:

- `A:shared [block_size*lda] type [n*lda]`
- `A:shared [block_size] type [n*lda]` in transpose case
- `C:shared [block_size*ldc] type [n*ldc]`

In addition, there are some limitations about the `block_size` parameter:

- The first element of `C` must be in the first row of a block in the shared memory space.
- The first element of `A` and `C` must be in parts of the shared memory with affinity to the same thread.
- If `A` non-transpose:
  - The first element of `A` must be in the first row of a block in the shared memory space.

- If `A` transpose:
    - The first element of `A` must be in the first column of a block in the shared memory space.
    - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $lda\%(block\_size * THREADS) == 0$.

2 `upc_pblas_colDist`: All the elements in a column of matrix `C` must have affinity to the same thread but the number of consecutive columns in each thread can change. In this case, this function treats array pointers as if they had type:

- `A:shared [block_size] type [n*lda]`
- `A:shared [block_size*lda] type [n*lda]` in transpose case
- `C:shared [block_size] type [n*ldc]`

In addition, there are some limitations about the `block_size` parameter:

- The first element of `C` must be in the first column of a block in the shared memory space.
- The first element of `A` and `C` must be in parts of the shared memory with affinity to the same thread.
- If `A` non-transpose:
    - The first element of `A` must be in the first column of a block in the shared memory space.
    - The first element of all rows must have affinity to thread 0 with phase 0. To achieve this, the following condition must hold: $lda\%(block\_size * THREADS) == 0$.
- If `A` transpose:
    - The first element of `A` must be in the first row of a block in the shared memory space.

If any array overlaps, the behavior is undefined.

Appendix B

Servet Library Interface

This appendix shows the syntax of the API of Servet, which is available in [106]. The Servet benchmarks write the hardware parameters in the file *config/config_system.txt* so that the users can easily access this information using the functions of this API. All programs that can work with C are able to deal with this interface.

The functions work with some datatypes, called descriptors, that store the information from the file. These descriptors were created as structures with different fields for the hardware parameters. Users do not need to know the name and meaning of each field as they do not need to access the fields directly: there are functions to provide hardware information from the descriptors.

B.1. Information about Cache Topology

All these functions are included in the file *cache.h* and use the descriptor `cache_desc` that keeps information about the cache topology.

B.1.1. load_cache_info

Function to load the information about the cache topology in the descriptor.

SYNTAX:

- `int load_cache_info(cache_desc *cache_info)`

PARAMETERS:

- `OUT cache_info`: Pointer to the descriptor where the information about the cache topology will be stored. It should not have been loaded before.
- returns:
  - 0 if everything is ok.
  - < 0 if an error occurs. The exact value is -j if an error is found in the jth field of `cache_desc`.

B.1.2. get_cache_nlevels

Function that provides the number of cache levels in the system.

SYNTAX:

- `int get_cache_nlevels(cache_desc *cache_info)`

PARAMETERS:

- `IN cache_info`: Pointer to the descriptor where the information about the cache topology is stored. It must have been loaded before.
- returns: Number of cache levels.

B.1.3. get_cache_size

Function that provides the size (in bytes) of a certain cache level.

SYNTAX:

- `int get_cache_size(cache_desc *cache_info, int level)`

PARAMETERS:

- `IN cache_info`: Pointer to the descriptor where the information about the cache topology is stored. It must have been loaded before.
- `IN level`: The selected cache level.
- returns:
  - The cache size (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.1.4. get_shared_cache_group_size

Function that provides the number of cores that share a certain cache level.

SYNTAX:

- `int get_shared_cache_group_size(cache_desc *cache_info, int level)`

PARAMETERS:

- `IN cache_info`: Pointer to the descriptor where the information about the cache topology is stored. It must have been loaded before.

- `IN level`: The selected cache level.
- returns:
  - The number of cores (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.1.5. get_shared_cache_cores

Function to obtain the set of cores that share a certain cache level with another core specified as parameter.

SYNTAX:

- `int get_shared_cache_cores(cache_desc *cache_info, int level, int core, int *shared_cores)`

PARAMETERS:

- `IN cache_info`: Pointer to the descriptor where the information about the cache topology is stored. It must have been loaded before.
- `IN level`: The selected cache level.
- `IN core`: The core that must be in the set of cores that share the cache level.
- `OUT shared_cores`: Output array where the set of cores that share the cache level with `core` are stored. It must have been allocated with enough space to store this data before calling the function.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.1.6. release_cache_info

Function to release all the information stored in the descriptor.

SYNTAX:

- `void release_cache_info(cache_desc *cache_info)`

PARAMETERS:

- `IN/OUT cache_info`: Pointer to the descriptor where the information about the cache topology is stored. It must have been loaded before.

B.2. Information about Shared Memory Overhead

All these functions are included in the file *mem_over.h* and use the descriptor `mem_over_desc` that keeps information about the shared memory access overhead.

B.2.1. load_mem_over_info

Function to load the information about the shared memory overhead in the descriptor.

SYNTAX:

- `int load_mem_over_info(mem_over_desc *mem_over_info)`

PARAMETERS:

- `OUT mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead will be stored. It should not have been loaded before.

- returns:
  - 0 if everything is ok.
  - < 0 if an error occurs. The exact value is -j if an error is found in the jth field of `mem_over_desc`.

B.2.2. get_mem_over_nlevels

Function that provides the number of different shared memory overhead levels in the system.

SYNTAX:

- `int get_mem_over_nlevels(mem_over_desc *mem_over_info)`

PARAMETERS:

- `IN mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead is stored. It must have been loaded before.
- returns: Number of shared memory overhead levels.

B.2.3. get_mem_over_mag

Function that provides the magnitude of a certain shared memory overhead level. The magnitude is measured as the percentage of the total memory bandwidth obtained when a pair of cores is accessing memory concurrently.

SYNTAX:

- `double get_mem_over_mag(mem_over_desc *mem_over_info, int level)`

PARAMETERS:

- `IN mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead is stored. It must have been loaded before.
- `IN level`: The selected shared memory overhead level.
- returns:
  - The magnitude (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.2.4. get_mem_over_group_size

Function that provides the number of cores that share a certain shared memory overhead level.

SYNTAX:

- `int get_mem_over_group_size(mem_over_desc *mem_over_info, int level)`

PARAMETERS:

- `IN mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead is stored. It must have been loaded before.
- `IN level`: The selected shared memory overhead level.
- returns:
  - The number of cores (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.2.5. get_mem_over_cores

Function to obtain the set of cores that share a certain shared memory overhead level with another core specified as parameter.

SYNTAX:

- `int get_mem_over_cores(mem_over_desc *mem_over_info, int level, int core, int *shared_cores)`

PARAMETERS:

- `IN mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead is stored. It must have been loaded before.
- `IN level`: The selected shared memory overhead level.
- `IN core`: The core that must be in the set of cores that share the memory overhead level.
- `OUT shared_cores`: Output array where the set of cores that share the memory overhead level with `core` are stored. It must have been allocated with enough space to store this data before calling the function.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.2.6. get_mem_over_group_mag

Function that provides the magnitude of a certain shared memory overhead level when a certain number of cores that share that level are accessing memory concurrently. The magnitude is measured as the percentage of the total memory bandwidth obtained.

SYNTAX:

- `double get_mem_over_group_mag(mem_over_desc *mem_over_info, int level, int ncores)`

PARAMETERS:

- `IN mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead is stored. It must have been loaded before.
- `IN level`: The selected shared memory overhead level.
- `IN ncores`: Number of cores of the shared memory level that access memory concurrently.
- returns:
  - The magnitude (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.2.7. release_mem_over_info

Function to release all the information stored in the descriptor.

SYNTAX:

- `void release_mem_over_info(mem_over_desc *mem_over_info)`

PARAMETERS:

- `IN/OUT mem_over_info`: Pointer to the descriptor where the information about the shared memory overhead is stored. It must have been loaded before.

B.3. Information about Communication Costs

All these functions are included in the file *comm.h* and use the descriptor `comm_desc` that keeps information about the communication costs.

B.3.1. load_comm_info

Function to load the information about communications in the descriptor.

SYNTAX:

- `int load_comm_info(comm_desc *comm_info)`

PARAMETERS:

- `OUT comm_info`: Pointer to the descriptor where the information about communications will be stored. It should not have been loaded before.
- returns:
  - 0 if everything is ok.
  - <0 if an error occurs. The exact value is -j if an error is found in the jth field of `comm_desc`.

B.3.2. get_comm_intra_node_nlevels

Function that provides the number of different communication layers within one node of the system.

SYNTAX:

- `int get_comm_intra_node_nlevels(comm_desc *comm_info)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- returns: Number of intra-node communication layers.

B.3.3. get_comm_intra_node_lat

Function that provides the latency (in milliseconds) of a certain intra-node communication layer when sending a message with size equal to the L1 cache size.

SYNTAX:

- `double get_comm_intra_node_lat(comm_desc *comm_info, int level)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- `IN level`: The selected intra-node communication layer.
- returns:
    - The latency (> 0) if everything is ok.
    - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.3.4. get_comm_inter_node_lat

Function that provides the latency (in milliseconds) of inter-node communications when sending a message with size equal to the L1 cache size.

SYNTAX:

- `double get_comm_inter_node_lat(comm_desc *comm_info)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- returns:
  - The latency (> 0) if everything is ok.
  - -100 if there is only one node in the system and thus there are no inter-node communications.
  - Another < 0 value if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.3.5. get_comm_intra_node_group_size

Function that provides the number of cores that share a certain intra-node communication layer.

SYNTAX:

- `int get_comm_intra_node_group_size(comm_desc *comm_info, int level)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- `IN level`: The selected intra-node communication layer.
- returns:
  - The number of cores (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.3.6. get_comm_intra_node_cores

Function to obtain the set of cores that share a certain intra-node communication layer with another core specified as parameter.

SYNTAX:

- `int get_comm_intra_node_cores(comm_desc *comm_info, int level, int core, int *shared_cores)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- `IN level`: The selected intra-node communication layer.
- `IN core`: The core that must be in the set of cores that share the intra-node communication layer.
- `OUT shared_cores`: Output array where the set of cores that share the intra-node communication layer with `core` are stored. It must have been allocated with enough space to store this data before calling the function.
- returns:
  - 0 if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.3.7. get_comm_min_msg_size

Function that provides the minimum message size (in bytes) used to study the communication bandwidths in all layers.

SYNTAX:

- `int get_comm_min_msg_size(comm_desc *comm_info)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- returns: The minimum message size.

B.3.8. get_comm_max_msg_size

Function that provides the maximum message size (in bytes) used to study the communication bandwidths in all layers.

SYNTAX:

- `int get_comm_max_msg_size(comm_desc *comm_info)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- returns: The maximum message size.

B.3.9. get_comm_intra_node_band

Function that provides the bandwidth (in MB/s) in a certain intra-node communication layer when sending a message with a certain size.

SYNTAX:

- `double get_comm_intra_node_band(comm_desc *comm_info, int level, int msg_size)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- `IN level`: The selected intra-node communication layer.
- `IN msg_size`: The selected message size.
- returns:
  - The bandwidth (> 0) if everything is ok.
  - < 0 if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.3.10. get_comm_inter_node_band

Function that provides the bandwidth (in MB/s) when sending an inter-node message with a certain size.

SYNTAX:

- `double get_comm_inter_node_band(comm_desc *comm_info, int msg_size)`

PARAMETERS:

- `IN comm_info`: Pointer to the descriptor where the information about communications is stored. It must have been loaded before.
- `IN msg_size`: The selected message size.
- returns:
  - The bandwidth (> 0) if everything is ok.
  - -100 if there is only one node in the system and thus there are no inter-node communications.
  - Another < 0 value if a parameter error occurs. The exact value is -j if the wrong parameter is the jth one.

B.3.11. release_comm_info

Function to release all the information stored in the descriptor.

SYNTAX:

- `void release_comm_info(comm_desc *comm_info)`

PARAMETERS:

- `IN/OUT comm_info`: Pointer to the descriptor where the information about the communications is stored. It must have been loaded before.

B.4. Process Mapping

The only function included in the file *automapping.h* obtains the appropriate process mapping. Two possible mapping policies are available:

- `SERVET_MEM_PRIOR`: It is focused on minimizing the overhead of sharing caches and memory. Secondarily, if possible, it also tries to improve latencies and bandwidths of communications.
- `SERVET_COMM_PRIOR`: It is focused on minimizing communication costs. Secondarily, if possible, it also tries to reduce the impact of concurrent shared memory accesses.

B.4.1. get_mapping_policy

Function that obtains the appropriate process mapping according to the hardware parameters detected by the benchmarks.

SYNTAX:

- `void get_mapping_policy(int np, int ncores, SERVET_PRIOR prior, int *policy)`

PARAMETERS:

- `IN np`: Number of processes that need to be mapped in the system.
- `IN ncores`: Total number of available cores. In a multicore cluster with n nodes and c cores per node where all cores are available for execution, the value of `ncores` must be $n * c$.
- `IN prior`: Identifier of the type of mapping policy that must be applied (`SERVET_MEM_PRIOR` or `SERVET_COMM_PRIOR`).
- `OUT policy`: Output array where the process mapping is stored. Entry i indicates the number of core where process i should be mapped. It must have been allocated with `np` elements before calling the function.