# Design of Scalable PGAS Collectives for NUMA and Manycore Systems

*Damián Álvarez Mallón*

Ph.D. in Information Technology Research

University of A Coruña, Spain

Ph.D. in Information Technology Research

University of A Coruña, Spain

Doctoral Thesis

# Design of Scalable PGAS Collectives for NUMA and Manycore Systems

Damián Álvarez Mallón

October 2014

PhD Advisor:
Guillermo López Taboada

Dr. Guillermo López Taboada
Profesor Contratado Doctor
Dpto. de Electrónica y Sistemas
Universidade da Coruña

CERTIFICA

Que la memoria titulada "*Design of Scalable PGAS Collectives for NUMA and Manycore Systems*" ha sido realizada por D. Damián Álvarez Mallón bajo mi dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña (UDC) y concluye la Tesis Doctoral que presenta para optar al grado de Doctor por la Universidade da Coruña con la Mención de Doctor Internacional.

En A Coruña, a Martes 10 de Junio de 2014

Fdo.: Guillermo López Taboada
Director de la Tesis Doctoral

*A todos/as os/as que me ensinaron algo*
*Polo bo e polo malo*

# Acknowledgments

Quoting Isaac Newton, but without comparing me with him: "*If I have seen further it is by standing on the shoulders of giants*". Specifically, one of the giants whose shoulder I have found particularly useful is my Ph.D. advisor, Guillermo López Taboada. His support and dedication are major factors behind this work, and it would not exist without them. I cannot forget Ramón Doallo and Juan Touriño, that allowed me become part of the Computer Architecture Group. Thanks to all of them I can call myself a member of the HPC community.

A special place in my acknowledgements is for my colleagues at the Galicia Supercomputing Center (CESGA). With the help of Andrés Gómez, Carlos Mouriño and Juan Pichel I have grow up professionally and gather a considerable amount of knowledge that just experience can provide. I cannot close this part of the acknowledgements without mentioning Brian Wibecan, that kindly hosted me at HP and shared with me his insightful ideas.

Lars Koesterke also has my gratitude. His support made possible the manycore evaluation, and his comments also improved it significantly.

I would like also to thank Wolfgang Gürich. He trusted me more than three years ago with a job at Jülich Supercomputing Centre (JSC). Without him a huge part of this Thesis would not be there, and I owe him much more than that. Thomas Fieseler also supported me during this endeavor, and for that I am grateful.

Of course I am also thankful to Hewlett-Packard S.L. for funding the project "Improving UPC Usability and Performance in Constellation Systems: Implementation/Extension of UPC Libraries", CESGA for kindly providing access to the Finis Terrae, Superdome and SVG systems, in particular the *sysadmins* Carlos, Javier, Pablo and Alex for all their patience, Texas Advance Computing Center (TACC) for the time allocated for the manycore evaluation in the Stampede supercomputer, and JSC for providing access to the JUDGE and JuRoPA clusters.

Last but not least –in fact quite the opposite–, I want to express my gratitude to my family. Their lessons are the most valuable that anybody can learn, and go well beyond what words can express.

*Damián Álvarez Mallón*

*Both optimists and pessimists contribute to our society.*
*The optimist invents the airplane and the pessimist the parachute.*
*Gil Stern*

# Resumo

O número de núcleos por procesador está crecendo, convertindo aos sistemas multinúcleo en omnipresentes. Isto implica lidiar con múltiples niveis de memoria en sistemas NUMA, accesibles a través de complexas xerarquías para procesar as crecentes cantidades de datos. A clave para un movemento eficiente e escalable de datos é o uso de operacións de comunicación colectivas que minimizen o impacto dos colos de botella. Usar comunicacións unilaterais vólvese máis importante nestes sistemas, para evitar sincronizacións entre pares de procesos en operacións colectivas implementadas usando funcións punto a punto bilaterais. Esta tese propón unha serie de algoritmos que proporcionan bo rendemento e escalabilidade en operacións colectivas. Estes algoritmos usan árbores xerárquicas, solapamento de comunicacións unilaterais, *pipelining* de mensaxes e afinidade NUMA. Desenvolveuse unha implementación para UPC, unha linguaxe PGAS cuxo rendemento tamén foi avaliado nesta tese. Para comprobar o rendemento destes algoritmos unha nova ferramenta de *microbenchmarking* foi deseñada e implementada. A avaliación dos algoritmos, realizada en 6 sistemas representativos, con 5 arquitecturas de procesador e 5 redes de interconexión diferentes, mostrou en xeral un bo rendemento e escalabilidade, mellor que os algoritmos líderes en MPI en moitos casos, o que confirma o potencial dos algoritmos desenvoltos para arquitecturas multi- e *manycore*.

# Resumen

El número de núcleos por procesador está creciendo, convirtiendo a los sistemas multinúcleo en omnipresentes. Esto implica lidiar con múltiples niveles de memoria en sistemas NUMA, accesibles a través de complejas jerarquías para procesar las crecientes cantidades de datos. La clave para un movimiento eficiente y escalable de datos es el uso de operaciones de comunicación colectivas que minimizen el impacto de los cuellos de botella. Usar comunicaciones unilaterales se vuelve más importante en estos sistemas, para evitar sincronizaciones entre pares de procesos en operaciones colectivas implementadas usando funciones punto a punto bilaterales. Esta tesis propone una serie de algoritmos que proporcionan buen rendimiento y escalabilidad en operaciones colectivas. Estos algoritmos usan árboles jerárquicos, solapamento de comunicaciones unilaterais, *pipelining* de mensajes y afinidad NUMA. Se ha desarrollado una implementación para UPC, un lenguaje PGAS cuyo rendimiento también ha sido evaluado en esta tesis. Para comprobar el rendimiento de estos algoritmos una nueva herramienta de *microbenchmarking* fue diseñada e implementada. La evaluación de los algoritmos, realizada en 6 sistemas representativos, con 5 arquitecturas de procesador y 5 redes de interconexión diferentes, ha mostrado en general un buen rendimiento y escalabilidad, mejor que los algoritmos líderes en MPI en muchos casos, lo que confirma el potencial de los algoritmos desarrollados para arquitecturas multi- y *manycore*.

# Abstract

The increasing number of cores per processor is turning multicore-based systems in pervasive. This involves dealing with multiple levels of memory in NUMA systems, accessible via complex interconnects in order to dispatch the increasing amount of data required. The key for efficient and scalable provision of data is the use of collective communication operations that minimize the impact of bottlenecks. Leveraging one-sided communications becomes more important in these systems, to avoid synchronization between pairs of processes in collective operations implemented using two-sided point to point functions. This Thesis proposes a series of collective algorithms that provide a good performance and scalability. They use hierarchical trees, overlapping one-sided communications, message pipelining and NUMA binding. An implementation has been developed for UPC, a PGAS language whose performance has been also assessed in this Thesis. In order to assess the performance of these algorithms a new microbenchmarking tool has been designed and implemented. The performance evaluation of the algorithms, conducted on 6 representative systems, with 5 different processor architectures and 5 different interconnect technologies, has shown generally good performance and scalability, outperforming leading MPI algorithms in many cases, which confirms the suitability of the developed algorithms for multi- and manycore architectures.

# Publications from the Thesis

## Journal Papers (3 accepted + 1 submitted)

- Damián A. Mallón, Guillermo L. Taboada and Lars Koesterke MPI and UPC Broadcast, Scatter and Gather Algorithms in Xeon Phi. Submitted to *Journal of Parallel Computing*, 2014

- Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Jorge González-Domínguez, Andrés Gómez and Brian Wibecan  Scalable PGAS Collective Operations in NUMA Clusters. In *Journal of Cluster Computing*, pages 1–23, 2014. Pending publication.

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, , Ramón Doallo, Damián A. Mallón, J. Carlos Mouriño, and Brian Wibecan. Design and Implementation of an Extended Collectives Library for Unified Parallel C. In *Journal of Computer Science and Technology*, volume 28, number 1, pages 72–89, 2012.

- Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada, Juan Touriño, Ramón Doallo, Damián A. Mallón, and Brian Wibecan. UPCBLAS: A Library for Parallel Matrix Computations in Unified Parallel C. In *Journal of Concurrency and Computation: Practice and Experience*, volume 24, number 14, pages 1645–1667, 2012.

## International Conferences (6)

- Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada, Juan Touriño, Ramón Doallo, Damián A. Mallón, and Brian Wibecan. A Library for Parallel Numerical Computation in UPC . Poster in *The 24th Conference on Supercomputing (SC'11)*, Seattle, WA, (USA), 2011.

- Damián A. Mallón, J. Carlos Mouriño, Andrés Gómez, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, and Brian Wibecan. UPC Operations Microbenchmarking Suite (Research Poster). Poster in *The 25th International Supercomputing Conference (ISC'10)*, Hamburg, Germany, 2010.

- Damián A. Mallón, J. Carlos Mouriño, Andrés Gómez, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, and Brian Wibecan. UPC Performance Evaluation on a Multicore System. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, pages 9:1–9:7, Ashburn, VA (USA), 2009.

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, Damián A. Mallón, Andrés Gómez, J. Carlos Mouriño, and Brian Wibecan. Evaluation of UPC Programmability Using Classroom Studies. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, pages 10:1–10:7, Ashburn, VA (USA), 2009.

- Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, José Carlos Mouriño Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th Euro PVM/MPI (EuroPVM/MPI'09)*, pages 174–184, Espoo, Finland, 2009.

- Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, José Carlos Mouriño, Damián A. Mallón, and Andrés Gómez. Performance Evaluation of Unified Parallel C Collective Communications. In *Proc. 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'09)*, pages 69-78, Seoul, Korea, 2009.

## Technical Reports (1)

- Damián A. Mallón, Andrés Gómez and Guillermo L. Taboada. Performance Evaluation of Finis Terrae. Technical Report CESGA-2009-001, 2009.

# Resumo da Tese

## Introdución

Nos últimos anos a mellora do rendemento dos procesadores tivo un cambio de tendencia. Tradicionalmente, con cada xeración de microprocesadores as aplicacións aumentaban o seu rendemento de forma totalmente transparente, debido ás melloras na microarquitectura e ao aumento de frecuencia. Isto xa non é posible hoxe, e os procesadores aumentan as súas prestacións doutro xeito. Nesta nova era os procesadores incrementan o seu número de núcleos. Ca aparición de coprocesadores *manycore* esta tendencia acentúase aínda máis, xa que o rendemento de cada núcleo individualmente non só non incrementa, senón que diminúe. Como efecto colateral deste feito, a presión sobre o subsistema de memoria é máis grande, e precísase maior ancho de banda a memoria. A resposta é a inclusión de varios controladores de memoria por nodo, o que causa que a maior parte dos sistemas de computación de altas prestacións modernos sexan Non-Uniform Memory Access (NUMA). A situación máis habitual é que cada procesador teña o seu propio controlador, aínda que nalgúns casos un único chip pode conter varios controladores de memoria.

Como resultado as aplicacións teñen que adaptarse acordemente. Seguindo a inercia inicial, o modelo de programación máis popular hoxe en día é un híbrido que usa paso de mensaxes (MPI) para comunicación entre procesos que non comparten memoria, e un modelo de fíos con memoria compartida (OpenMP). Porén, ese modelo dificulta a programación das aplicacións, e é proclive a erros. A comunidade de computación de altas prestacións propuxo unha alternativa, denominada Partitioned Global Address Space (PGAS). Neste modelo hai unha parte da memoria que é lóxicamente compartida, aínda que físicamente non o sexa. Deste xeito evítanse os problemas e a complexidade que o modelo de paso de mensaxes trae consigo, e permítense comunicacións unilaterais que non requiren sincronización. No entanto, para obter bo rendemento cómpre ter unha boa localidade de datos, xa que acceder á memoria remota constantemente resultaría nun mal rendemento.

En todo modelo de programación que inclúa memoria distribuída a algún nivel é necesario ter primitivas eficientes de redistribución de datos. Estas primitivas son operacións colectivas executadas por todos os procesos que participan na devandita operación. A maioría das aplicacións usan estas operacións colectivas nalgún punto da súa execución. Por tanto, a escalabilidade e rendemento destas operacións xoga un papel moi importante para permitir mellores tempos de execución en situacións con decenas de miles de núcleos.

A confluencia de todos estes factores –novas arquitecturas de procesadores con moitos máis núcleos que as tradicionais e xerarquías de memoria máis profundas, novos modelos de programación, e a necesidade de operacións de redistribución de datos eficientes e escalables– é a motivación desta tese de doutoramento, titulada "Design of Scalable PGAS Collectives for NUMA and Manycore Systems". O traballo focalizouse en Unified Parallel C (UPC), unha linguaxe PGAS baseada en C, e unha das alternativas PGAS máis coñecidas. Co fin de probar o potencial de UPC para a computación de altas prestacións nesta tese se realizou unha avaliación de rendemento desta linguaxe, co compilador e runtime de UPC máis popular (Berkeley UPC). Adicionalmente, co obxetivo de avaliar o rendemento dos algoritmos propostos, desenvolveuse unha ferramenta de medición de rendemento, que constitúe a primera do seu xénero en UPC. O último paso foron avaliacións de rendemento dos algoritmos en múltiples contornas, empregando 5 sistemas NUMA e 1 sistema *manycore*.

## Metodoloxía de Traballo

A metodoloxía de traballo desta tese consta dunha serie de tarefas para realizar. Tales tareas deberán:

- Ter en conta o estado da arte e os recursos cos que se contan.

- Ser organizadas en certa orde lóxica.
- Ter unha duración determinada.
- Formar bloques de traballo relacionados, de forma que cada bloque represente unha etapa claramente distinguible.
- Ter unha serie de metas por bloque que determinan o éxito da tese.

Ademais, dada a rápida evolución da computación de altas prestacións, novas tarefas pódense levar a cabo se os recursos dispoñibles o permiten e os resultados obtidos teñen un impacto significativo. Deste xeito, a listaxe de tarefas (*Tn*), agrupadas en bloques (**Bn**), desenvolvidas na presente tese foron:

**B1** Estudo do estado da arte en modelos de programación e algoritmos de operacións colectivas.

- *T1.1* Estudo de alternativas actuais para programación paralela, incluíndo modelos de programación para sistemas de memoria distribuída e memoria compartida.
- *T1.2* Familiarización coas linguaxes existentes que foron estendidas para soportar características PGAS, linguaxes creadas especificamente, e outras alternativas PGAS baseadas en bibliotecas.
- *T1.3* Estudo de características relevantes da linguaxe UPC, con atención ás características útiles desde o punto de vista de deseño de operacións colectivas.
- *T1.4* Estudo en profundidade de algoritmos para operacións colectivas na literatura, con especial atención ao estado da arte á referencia en HPC, MPI.

**B2** Estudo de rendemento de UPC.

- *T2.1* Busca de ferramentas de avaliación de rendemento que permitan comparación entre MPI –por ser a alternativa de programación paralela máis popular– e UPC.

- *T2.2* Avaliación de rendemento de UPC cos devanditos *benchmarks*, incluíndo unha análise comparativa con MPI.
- *T2.3* Análise doutros traballos de avaliación de rendemento de UPC, con especial atención ás súas conclusións sobre o posible futuro de UPC.

**B3** Deseño e implementación dunha ferramenta de medición de rendemento.

- *T3.1* Estudo de alternativas de *microbenchmarking* para MPI e OpenMP en distintos eidos, en particular no tocante a comunicacións e operacións colectivas.
- *T3.2* Deseño dunha ferramenta de *microbenchmarking* de comunicacións en UPC, que resulte familiar a outros usuarios con experiencia en MPI e que permita realizar estudos comparativos.
- *T3.3* Implementación da primera ferramenta de *microbenchmarking* de comunicacións en UPC.

**B4** Deseño e implementación de algoritmos para operacións colectivas para arquitecturas NUMA e multicore.

- *T4.1* Deseño dun algoritmo xeral que sirva de base para varias operacións colectivas, baseado nunha estructura de árbore, e que aproveite as operacións unilaterais de UPC.
- *T4.2* Exploración do espazo de posibles optimizacións do algoritmo xeral, incluíndo solapamento de comunicacións, distintas formas de árbore, e técnicas para obter escalabilidade con miles de núcleos.
- *T4.3* Refinamento e adaptación do algoritmo para as operacións que así o requiran debido as súas particularidades, no tocante á distribución de datos e operacións relacionadas.
- *T4.4* Implementación da familia de algoritmos na súa forma básica, para usar como orixe das versións optimizadas.
- *T4.5* Implementación usando os algoritmos base das optimizacións deseñadas de forma incremental.
- *T4.6* Estudo de optimizacións extra para funcións que non encaixen no algoritmo xeral debido aos movementos de datos necesarios.

*T4.7* Implementación das optimizacións para ditas funcións.

**B5** Análise de rendemento dos algoritmos implementados en arquitecturas NUMA.

*T5.1* Avaliación de rendemento dos algoritmos baseados en árbores en múltiples sistemas NUMA con diferentes características –procesadores, interconexión intranodo e interconexión internodo–.

*T5.2* Análise comparativa con outras alternativas en UPC, en particular a implementación de referencia e a implementación do *runtime* con máis rendemento dispoñible.

*T5.3* Análise comparativa co estado do arte en MPI en experimentos a grande escala con miles de núcleos.

*T5.4* Análise da contribución de cada técnica de optimización ao rendemento final de cada variación das operacións colectivas.

**B6** Análise de rendemento dos algoritmos implementados en arquitecturas *manycore.*

*T6.1* Avaliación de rendemento dos algoritmos baseados en árbores en sistemas *manycore*, escalando os experimentos ata a orde de decenas de miles de núcleos.

*T6.2* Análise comparativa co estado da arte en MPI en ditos sistemas, incluíndo runtimes especialmente adaptados e optimizados para dita arquitectura.

*T6.3* Análise comparativa do rendemento das colectivas en UPC e MPI en procesadores *manycore* –Intel Xeon Phi– contra o rendemento obtido en procesadores *multicore.*

**B7** Extracción de conclusións.

*T7.1* Resumo do traballo e extracción de conclusións.

*T7.2* Análise de traballo futuro e liñas abertas.

*T7.3* Escritura da memoria da tese de doutoramento.

A presente memoria recolle o traballo feito en cada tarea. Cada capítulo contén exactamente un bloque dos anteriormente enumerados.

Cada bloque ten unha serie de metas que se queren alcanzar. A listaxe de metas (*Mn*) asociadas con cada bloque (**Bn**) da tese de doutoramento foi:

**B1** Estudo do estado da arte en modelos de programación e algoritmos de operacións colectivas.

- *M1.1* Obtención de perspectiva en canto a opcións dispoñibles para programación paralela de hardware moderno.
- *M1.2* Obtención de ideas para o deseño e implementación de operacións colectivas escalables ata miles de núcleos.

**B2** Estudo de rendemento de UPC.

- *M2.1* Determinación da validez de UPC como alternativa para programación de sistemas paralelos para aplicacións científicas.

**B3** Deseño e implementación dunha ferramenta de medición de rendemento.

- *M3.1* Proporcionar á comunidade da primeira ferramenta de *microbenchmarking* de comunicacións para UPC.

**B4** Deseño e implementación de algoritmos para operacións colectivas para arquitecturas NUMA e multicore.

- *M4.1* Conxunto de rutinas de operacións colectivas en UPC altamente optimizadas e escalables a sistemas con decenas de miles de núcleos e múltiples rexións NUMA por nodo.

**B5** Análise de rendemento dos algoritmos implementados en arquitecturas NUMA.

- *M5.1* Comprensión do rendemento e escalabilidade das operacións optimizadas comparativamente co estado da arte actual en UPC, en sistemas NUMA.
- *M5.2* Comprensión do rendemento e escalabilidade das operacións optimizadas comparativamente co estado da arte actual en MPI, en sistemas NUMA.

- *M5.3* Comprensión do impacto no rendemento e escalabilidade das distintas técnicas de optimización implementadas.

**B6** Análise de rendemento dos algoritmos implementados en arquitecturas *many-core.*

- *M6.1* Comprensión do rendemento e escalabilidade das operacións optimizadas comparativamente co estado da arte actual en MPI, en sistemas *many-core.*
- *M6.2* Comprensión do impacto de cambios profundos na arquitectura de procesadores no rendemento e escalabilidade de operacións colectivas.

**B7** Extracción de conclusións.

- *M7.1* Memoria da tese de doutoramento, coa descrición dos pasos realizados e as conclusións obtidas, así como as futuras liñas de traballo.

Os medios necesarios para realizar esta tese de doutoramento, seguindo a metodoloxía de traballo anteriormente descrita, foron os seguintes:

- Proxecto de investigación con financiamento privado e de ámbito internacional: "Improving UPC Usability and Performance in Constellations Systems: Implementation/Extension of UPC Libraries". Financiado por Hewlett-Packard S.L., en colaboración coa Universidade da Coruña, a Universidade de Santiago de Compostela e o Centro de Supercomputación de Galicia.
- Clústers utilizados durante o desenvolvemento e avaliación de rendemento dos algoritmos desta tese:
    - *Supercomputador Finis Terrae* (Centro de Supercomputación de Galicia, 2008-2011): 142 nodos con 8 procesadores Intel Itanium 2 dual core, a 1.6 GHz e 128 GB de RAM. A rede de interconexión deste supercomputador é InfiniBand 4x DDR.
    - *Nodo Superdome* (Centro de Supercomputación de Galicia, 2008-2011): Nodo de cómputo con 64 procesadores Intel Itanium 2 dual core, a 1.6 GHz e 1 TB de RAM.

  - *Supercomputador Virtual Galego 2011 (SVG 2011)* (Centro de Supercomputación de Galicia, 2011): 46 nodos con 2 procesadores AMD Opteron, cada un con 12 cores, a 2.2 GHz e 64 GB de RAM. A rede de interconexión é Gigabit Ethernet.
  - *JUDGE* (Forschungzentrum Jülich, 2011-actualidade): 206 nodos con 2 procesadores Intel Xeon hexa core, a 2.66 GHz e 96 GB de RAM. A rede de interconexión é InfiniBand 4x QDR.
  - *JuRoPA* (Forschungzentrum Jülich, 2011-actualidade): 2208 nodos con 2 procesadores Intel Xeon quad core, a 2.93 GHz e 24 GB de RAM. A rede de interconexión é InfiniBand 4x QDR.
  - *Stampede* (Texas Advanced Computing Center, 2013-actualidade): 6400 nodos con 2 procesadores Intel Xeon octo core, a 2.7 GHz e 32 GB de RAM. Cada nodo conta, ademais, con 1 coprocesador Intel Xeon Phi con 61 cores a 1.1 GHz e 8 GB de RAM. A rede de interconexión é InfiniBand 4x FDR.

- Contrato con Forschungszentrum Jülich GmbH, que excede no momento de escribir esta Tese os 3 anos. Dito contrato posibilitou a realización das tarefas incluídas nos Bloques 5 e 6.

## Conclusións

Esta tese de doutoramento, "*Design of Scalable PGAS Collectives for NUMA and Manycore Systems*", demostrou o potencial do modelo PGAS para a implementación de operacións colectivas eficientes e altamente escalables, chegando en moitos casos a render mellor que as actuais solucións neste eido, en moitos escenarios diferentes. A programación de sistemas cada vez máis complexos, con memoria distribuída e compartida, con distintas latencias de acceso a memoria dependendo de que dirección se esté accedendo, e cun hardware evolucionando a alta velocidade, é altamente complicada. O modelo PGAS propón unha alternativa ao tradicional modelo de paso de mensaxes, con algunhas características que evitan en certa medida custosas sincronizacións, e que se poden aplicar ao deseño de bibliotecas de comunicacións.

Os programadores de aplicacións científicas usadas en sistemas de altas prestacións recorren con frecuencia a operacións colectivas incluídas nos *runtimes* correspondentes. Como consecuencia, os enxeñeiros de computadores levan anos optimizando e adaptando tales operacións a distintas arquitecturas, dando como resultado unha extensa traxectoria de investigación e historial de algoritmos. A pesar desta traxectoria, o traballo que explora a optimización das ditas operacións usando o modelo PGAS é escaso.

Esta tese analizou o rendemento de UPC en *benchmarks* científicos, concluíndo que é comparable ao rendemento doutras aplicacións baseadas en C e que usan paso de mensaxes como método de comunicación. Porén, e a pesar da boa escalabilidade nalgúns casos, tamén se achou que cómpre prestar atención á optimización do acceso a rede, que nalgúns casos causa unha perda de rendemento usando multiples núcleos. Isto sinala a importancia de optimizar as operacións de comunicacións. En particular, as operacións colectivas teñen especial importancia, debido a súa popularidade e uso frecuente.

No tocante ao deseño de colectivas, esta tese non podía efectuar un traballo metódico e exhaustivo sen contar con ferramentas para medir o rendemento das ditas operacións. Ante a ausencia de ferramentas axeitadas deseñouse e implementouse a primeira ferramenta de *microbenchmarking* para UPC, usando o coñecemento e experiencia obtidos durante anos por científicos, usando paso de mensaxes. Con esta ferramenta fundamental implementada, a avaliación de rendemento en colectivas xa é posible.

No núcleo desta tese está o deseño de algoritmos altamente escalables para operacións colectivas en arquitecturas modernas de supercomputación. Tales algoritmos, implementados en UPC, apoianse nas seguintas técnicas para obter un alto rendemento e escalabilidade:

- Uso de operacións de comunicación unilaterais, empuxando ou tirando de datos –é dicir, a orixe da operación está no proceso que ten ou necesita os datos, respectivamente–, dependendo da natureza da operación.

- Uso de árbores cunha estrutura computada na inicialización e reusada durante todo o tempo de execución da aplicación.

- Árbores xerárquicas mapeadas eficientemente no *hardware*, minimizando o uso dos camiños con latencia máis alta e anchos de banda máis pequenos.

- Distinto tipo de árbores no nivel máis baixo, permitindo elixir a forma máis axeitada en cada caso.

- *Binding* de fíos a rexións NUMA, para asegurar o correcto mapeamento das árbores.

- Dúas técnicas de solapamento de comunicacións, usando tamaños fixos e dinámicos.

As operacións optimizadas avaliáronse exahustivamente en 6 supercomputadores distintos (Stampede, JuRoPA, JUDGE, Finis Terrae, SVG e Superdome), con 5 arquitecturas de procesador distintas (Intel Xeon Phi Many Interconnected Core, Intel Xeon Sandy Bridge, Intel Xeon Nehalem, Intel Itanium 2 e AMD Opteron Magny-Cours), e 5 tecnoloxías de interconexión diferentes (InfiniBand 4x FDR, InfiniBand 4x QDR, InfiniBand 4x DDR, Gigabit Ethernet e Superdome Interconnect). O rendemento e a escalabilidade dos algoritmos deseñados sobrepasa ao das mellores colectivas avaliadas en UPC en case todos os escenarios. As colectivas presentes en implementacións punteiras de *runtimes* MPI tamén se vén sobrepasadas con frecuencia polos algoritmos desta tese, principalmente en escenarios cun alto número de núcleos. Especialmente destacable é o bo rendemento e escalabilidade en sistemas *manycore*, usando máis de 15000 núcleos, a pesar de que o rendemento no tocante a comunicacións nestos sistemas é xeralmente inferior ao obtido en sistemas tradicionais multinúcleo.

## Principais Contribucións

As principais achegas desta tese son:

- Un estudo de rendemento das operacións colectivas en UPC, previamente aos algoritmos presentados, e materializado en [100].

- Un estudo de rendemento de UPC en *benchmarks* científicos de gran relevancia, e unha análise comparativa con outras opcións populares –MPI e OpenMP–, publicada en [58] e [63].
- A primera ferramenta de *microbenchmarking* para comunicacións en UPC, dispoñible públicamente en [60] e que a día de hoxe acumula máis de 370 descargas. Tal ferramenta foi presentada en [59].
- Unha biblioteca con operacións colectivas altamente optimizadas para arquitecturas modernas usando diversas técnicas.
- Un estudo da devandita biblioteca en 5 sistemas NUMA diferentes (JuRoPA, JUDGE, Finis Terrae, SVG e Superdome), no que se demostra un gran rendemento, incluso comparado co equivalente ao estado do arte en MPI, e publicado en [62].
- Un estudo dos algoritmos deseñados en un dos sistema *manycore* máis potentes do mundo neste momento (Stampede), no que os algoritmos desta tese teñen un desempeño que sobrepasa en moitos escenarios ao de algoritmos implementados en *runtimes* máis optimizados para dito sistema, traballo do que se derivou unha publicación que está sendo considerada para a súa publicación [61].

# Contents

# List of Algorithms

# List of Tables

# List of Figures

# Preface

Processor manufacturing hit the power wall nearly 10 years ago. As a result the microprocessor industry was forced to embrace a new era where the performance for new generation microprocessors came mainly from exposing more parallelism, rather than increased frequency. This posed a major shift in the industry, whose consequences are still difficult to deal with today. As a result current computer systems are based on multicore chips, which are constantly increasing their number of cores. This scenario, and particularly the new multicore processor architectures, heightens the importance of memory performance and scalability. The inclusion of the memory controller inside the processor's chip helps to minimize the issues associated with the access to shared memory from multicore chips. As a result, currently most systems, both single socket and multi-socket, have Non Uniform Memory Access (NUMA) architectures, as now processors package several memory controllers in a single chip. NUMA architectures provide scalability through the replication of paths to main memory, reducing the memory bus congestion as long as the accesses are evenly distributed among all the memory modules. This scalability is key for applications running on thousands of cores. The supercomputing community is now hitting a new power wall, where the power budget for supercomputers cannot grow anymore. As a result new creative ways are needed to circumvent this limitation. New approaches propose using small low-power processors like the ones used in cell phones (ARM) [92], with promising results. However, nowadays solutions rely in power-hungry manycore microarchitectures, such as Graphic Processing Units (GPUs) or Many Integrated Core (MICs) coprocessors, but with a good MFLOPS/Watt ratio. The emergence of these manycore devices has caused major changes in the HPC community. It has introduced a new level of complexity that the application developers have to deal with, adapting their applications to the new hardware, the new levels of memory hierarchy present in the new processors, and the new APIs.

As a result of this ever increasing complexity, the HPC community has proposed new ways to program current systems. In the last few years a new programming model, Partitioned Global Address Space (PGAS), is becoming more popular. Despite not having the relevance of MPI, different important research groups are focusing on PGAS languages due to their potential [20]. One of their main features, one-sided asynchronous memory copies, has been already adopted in MPI. This feature can have an important role in the design of the ever important collective operations in scenarios with thousands of cores, where synchronizations becomes especially costly. Given this confluence of factors –complex hardware architectures with deeper memory hierarchies, new programming models and the importance to have efficient and scalable collective operations– this Thesis has designed, implemented and evaluated a set of collective operations in Unified Parallel C (UPC) –a PGAS language–.

Evaluation of performance is a central task in the development of HPC libraries of any kind. However, focusing on collectives for UPC, there is a lack of tools to assess methodically the performance of these libraries. As a prerequisite for the design and implementation of highly scalable collective algorithms, such a tool has been developed as part of this work, constituting one of the major contributions of this Thesis, as it provides a reference tool, equivalent to the Intel MPI Benchmarks for MPI runtimes.

The contents of this Ph.D. Thesis are organized as follows:

- Chapter 1, *Background and State-of-the-art in Collective Operations* provides the reader with an insightful explanation of the issues of programming modern high performance computing hardware, and how this affects the programming models used in this field. It continues with a description of the PGAS programming model, and an introduction of UPC as a popular PGAS language. Lastly, it provides a complete explanation of the state-of-the-art algorithms for collective operations.

- Chapter 2, *Characterization of UPC Performance* analyzes the performance of UPC in popular benchmarks, assessing its suitability for high performance computing and postulating it as a candidate for the programming of current and future systems.

- Chapter 3, *UPC Operations Microbenchmarking Suite: UOMS* describes the design and implementation of the first microbenchmarking suite for UPC, explaining all the benchmarks included in the suite and various options available.

- Chapter 4, *Design of Scalable PGAS Collective Algorithms* details the design of the proposed PGAS collectives, and all the optimizations implemented in them. However, introductorily, it describes the existing collectives in UPC. Given that not all the operations distribute data equally, implementation details are discussed, with particular emphasis on collectives that fit tree structures.

- Chapter 5, *Perf. Evaluation of PGAS Collectives on NUMA Systems* analyzes the performance of the proposed algorithms for broadcast, reduce, scatter and gather in 5 different NUMA architectures. Comparisons with the reference implementation, and with the state-of-the-art collective library in UPC have been made. Additionally, the performance of a leading MPI implementation is compared with the performance of the proposed algorithms, and the impact of the different optimizations has been assessed and explained here.

- Chapter 6, *Perf. Evaluation of PGAS Collectives on Manycore Systems* analyzes the performance of the proposed algorithms for broadcast, scatter and gather in a manycore system, paying special attention to how they compare with the best of class algorithms for MPI for this platform, including manycore-optimized MPI runtimes.

- *Conclusions and Future Work* closes this Thesis, summarizing the conclusions and describing future lines of work.

Additionally, the user manual of the UPC Operations Microbenchmarking Suite is included as an appendix.

# Chapter 1

# Background and State-of-the-art in Collective Operations

The hardware used in current supercomputers has been following a trend since it hit the power wall, where the increase in performance came from exposing more parallelism –i.e. implementing multicore chips– instead of developing larger cores with higher clock speeds [49]. This poses a significant change, since now applications do not get a performance boost just by simply replacing the processor. Now parallelism at various levels becomes indispensable to achieve performance. In order to help programmers to cope with this new challenge, different programming models have been proposed. These new programming models open the opportunity to develop algorithms in more expressive ways. Collective operations, traditionally a key part of message-passing software, due to its convenience and optimized algorithms, can benefit from some of the features proposed by these new programming models.

This Chapter briefly discusses the most relevant programming models for High Performance Computing (HPC) in Section 1.1. Sections 1.2 and 1.3 complement the previous Section providing an overview of the Parallel Global Address Space (PGAS) programming model and its incarnations, as well as an introduction to Unified Parallel C (UPC). Section 1.4 provides a deep explanation of current state-of-the-art collective algorithms. Finally, Section 1.5 concludes the Chapter and discusses how PGAS features bring new opportunities for optimization of collective operations.

## 1.1. Programming Models

In the current multipetascale era, the discussion about programming models is becoming increasingly important. The complex modern hardware architectures require the use of multiple levels of paralellization. For internode parallelization, the message-passing is the most widely used parallel programming model as it is portable, scalable and provides good performance for a wide variety of computing platforms and codes. As the programmer has to manage explicitly data placement through point-to-point or collective operations, the programming of message passing software is difficult. MPI is the standard interface for message-passing libraries and there is a wide range of MPI implementations, both from HPC vendors and the free software community, optimized for high-speed networks, such as InfiniBand or Myrinet. MPI, although is oriented towards distributed memory environments, faces the raise of the number of cores per system with the development of efficient shared memory transfers and providing thread safety support.

For the intranode parallelization OpenMP is the most widely used solution, as it allows an easy development of parallel applications through compiler directives, that mainly distribute the work to be done between a number of threads. Moreover, it is becoming more important as the number of cores per system increases. However, as this model is limited to shared memory architectures, the performance is bound to the computational power of a single system. To avoid this limitation, hybrid systems, with both shared/distributed memory, such as multicore clusters, can be programmed using MPI+OpenMP. However, this hybrid model can make the parallelization more difficult and the performance gains could not compensate for the effort [89, 90]. In the last incarnation of OpenMP [83] –4.0 heavily influenced by OmpSs [17]– the concept of tasks becomes more important, as dependencies between tasks can be specified, allowing the runtime to create a graph that exposes more parallelism and hence increase efficiency of multi- and manycore systems.

In the last years, modern HPC systems have incorporated accelerators to boost computational power. This has introduced yet another level of complexity that requires programmers to deal with another programming model. This way, CUDA [74] (for NVIDIA accelerators) and OpenCL [99] (for NVIDIA and AMD accelerators) implement programming models focused on the efficient exploitation of accelerators with massive amounts of concurrent threads. Intel has also released their MIC – Many Integrated Core– architecture [41], based on x86 processors, that incorporates up to 61 cores in a coprocessor with very wide vector units, whose target is to provide a significant performance boost without requiring a complete rewriting of the applications.

The PGAS programming model combines some of the main features of the message-passing and the shared memory programming models. In PGAS languages, each thread has its own private memory space, as well as an associated shared memory region of the global address space that can be accessed by other threads, although at a higher cost than a local access. Thus, PGAS languages allow shared memory-like programming on distributed memory systems. Moreover, as in message-passing, PGAS languages allow the exploitation of data locality as the shared memory is partitioned among the threads in regions, each one with affinity to the corresponding thread.

## 1.2. PGAS Programming Model

The PGAS programming model has its origins in the appearance of the SHMEM (Symmetric Hierarchical Memory) [3] and Global Arrays libraries [75, 76, 85], around 1994. SHMEM is a family of libraries initially developed by Cray for its T3D supercomputer, that provides a shared-memory-like API to access memory in distributed memory systems. Likewise, Global Arrays is a toolkit developed by Pacific Northwest National Laboratory, that also provides a shared-memory-like API, and was developed with portability and efficiency in mind.

Acknowledging the potential of these libraries, a series of extensions were proposed to popular languages, in order to provide PGAS features to them. This way, Co-Array Fortran (CAF) [79, 80] is based on Fortran 95, adding PGAS support to it. CAF was included in the Fortran 2008 standard, becoming now an important part of the Fortran language. Unified Parallel C (UPC) [114] is based in C99, and adds support for declaring shared variables among UPC threads, as well as its distribution. It also provides a set of libraries. Titanium [121] is a PGAS extension to Java, whose code normally is compiled to native code instead of the byte code that runs in Java Virtual Machines.

Besides these extensions, the Defense Advanced Research Projects Agency (DARPA) of the United States of America developed the High Productivity Computing Systems (HPCS) programme, resulting in funding for the development of high productivity parallel languages. Each of these languages have been designed from scratch, incorporate PGAS features, and has been developed by a different vendor. X10 [10, 36] is an object oriented language developed by IBM, with two levels of concurrency, one for shared memory environments and another one for distributed memory, mapping accordingly to a cluster architecture. Chapel [7, 13] has been developed by Cray, with data and task parallelization as the main focus. Fortress [84, 98] is the Sun Microsystems (now Oracle) proposal for the HPCS, with implicit parallel features and advanced mathematical notation.

In the last few years, the library approach to PGAS has been also materialized basically in two libraries. OpenSHMEM [9] is the standard promoted after SHMEM, to ensure portability and avoid vendor lock in. Global Address Space Programming Interface (GASPI) [23] is another promising library, that allows to define different memory segments and provides advanced PGAS features.

Despite their differences, all these PGAS languages and libraries provide the programmer with the possibility of accessing remote memory directly, without intervention of the process whose memory is being accessed.

## 1.3. Introduction to UPC

UPC is the PGAS extension to the ISO C99 language, and has been used in this evaluation due to its important support by academia and industry. UPC provides PGAS features to C, allowing a more productive code development than other alternatives like MPI [19, 105]. Due to its programmability and performance –as well as being an extension of a popular programming language–, UPC has being in the focus of the research community for some time. There are commercial –IBM, Cray, HP– implementations, as well as more open initiatives –from the University of California Berkeley, Michigan Technological University, Intrepid/GCC, Ohio State University–, and optimized runtimes are available for all the major –and some minor, like the Tile64 processor [96]– platforms [35, 51, 56]. Recently, even GPU support was proposed and implemented [11], underlying the importance of UPC. However, as an emerging programming model, performance analysis are needed [6, 18, 22].

In the memory model defined by UPC, there are two very well differentiated regions: private and shared. The private region of a UPC thread is the same as a private region of a process. That is, belongs exclusively to that UPC thread, and no other UPC thread can directly access it. The shared region is the memory part that is readable and writeable by all the UPC threads in the job. However, this shared memory is divided in blocks, each of which has affinity –hence faster access– to one UPC thread.

The programmer can define any kind of variable to be either private –by default– or shared, using the `shared` qualifier. In the case of defining shared arrays, the programmer can specify the blocking factor. I.e.: How many contiguous elements of the array belong to each thread. There are 4 ways to specify the blocking factor:

- Avoiding the use of the blocking factor: `shared int A[N]`. Implies a distribution with a block factor of 1.
- Using an empty blocking factor: `shared [] int A[N]`. Implies that the array is stored completely in the shared memory with affinity to the thread 0, i.e. a blocking factor of N.
- Using an explicit blocking factor: `shared [2] int A[N]`. Implies that the block factor is 2, i.e.: any given thread will hold in its shared memory 2 consecutive elements of the array.
- Using an automatic blocking factor: `shared [*] int A[N]`. In this case the runtime will determine the blocking factor, trying to use a blocking factor as big as possible, a fair distribution among threads but without allocating more than one block per thread. I.e. a blocking factor of $\lfloor(N + THREADS - 1)/THREADS\rfloor$.

Figure 1.1 illustrates how the different blocking factors affect the distribution of arrays in UPC.

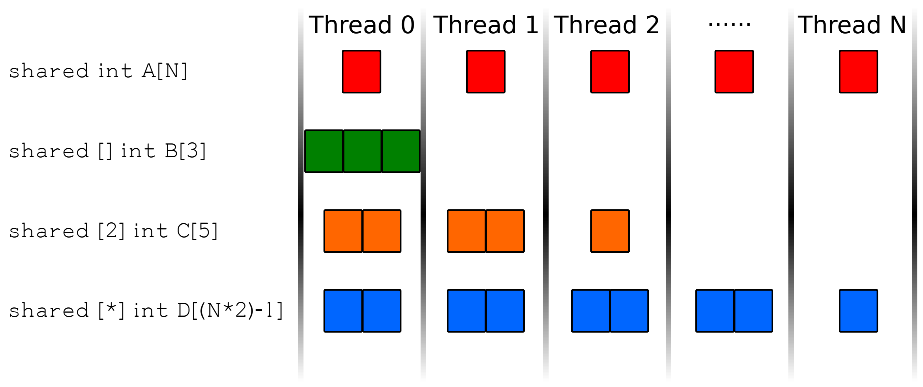


Figure 1.1: Distribution of arrays in shared memory in UPC

Memory, as in C, can be accessed with pointers. However, the addition of the shared memory space creates 4 possible situations:

- Private pointer to private memory. These are the regular C pointers.
- Private pointer to shared memory. These pointers allow UPC threads to keep references to shared memory in their private memory.
- Shared pointer to private memory. These pointers should be avoided. The pointers to private memory regions should not be allocated in the shared region, as it can lead to wrong memory accesses, due to potentially different private memory addresses of each variable.
- Shared pointer to shared memory. These pointers allow to share references to shared memory regions among different UPC threads.

Access to shared memory can be done just by referencing the desired variable, regardless of which threads owns that particular variable. However, there are a set of memory operations provided in order to perform copies of multiple elements at once. UPC initially provided `upc_memcpy`, `upc_memget` and `upc_memput`. `upc_memcpy` copies data from one shared memory location to another shared memory location. `upc_memget` and `upc_memput` operate similarly, but copying data from/to shared memory into/from private memory. This allows implicit parallelism and overlapping when distributing data, as multiple operations can be initiated simultaneously, one from each thread. This is of upmost importance in the design of PGAS collective operations, as explained in Chapter 4, resulting in two variants: Pull and push, depending on the nature of the collective.

In the last UPC specification, 1.3 [115, 116, 117], there are also the non-blocking variants of these operations. The non-blocking variants have been proposed by the University of California Berkeley, and have been implemented in Berkeley UPC [51] before the last UPC specification.

The one-sided memory operations provided by UPC are a natural way of moving data in this language. Due to its asynchronicity, were UPC threads involved –source and destination– do not need to agree in establishing a communication, they are a powerful foundation for the implementation of collective operations.

## 1.4. Collective Operations

The optimization of middleware for HPC is a complex task. It might involve all the layers of the runtime, and it should evolve with new versions of the language and library APIs [42, 122]. Within the runtime optimization, one of the most important points is the optimization of the collective operations, as most applications rely on them, both for programmability, as they implement popular operations, relieving the programmer from its error prone implementation, and also for performance, as they generally implement optimized and refined algorithms. Collective operations are usually key to achieve a good scalability. There are basically two approaches for the optimization of collective operations: the algorithmic and the system approach.

The algorithmic approach focuses on how the data is transferred and how the processes are organized. Not all algorithms can be suitable or implemented for all systems. Previous works on this field can be split between two main groups: distributed memory algorithms and shared memory algorithms, which target the main current architectures.

Kandalla et al. [46] developed and tested a topology-aware algorithm that builds the interconnect tree on InfiniBand clusters taking into account the process placement in relation to the switches, avoiding unnecessary switch hops. Similarly, Gong et al. developed an algorithm for MPI collectives in the cloud [28]. Bibo Tu et al.

[113] described a new broadcast algorithm for multicore clusters. In this algorithm two sets of communicators were used. The first one for intranode communications, where binding is used to improve locality within a node. The second one is for internode communications. This way a broadcast is performed in two steps, inter- and intranode transfer steps, avoiding the network interface congestion. Kumar et al. [50] designed and evaluated an all-to-all algorithm for multicore clusters. This algorithm is similar to the Bibo Tu's broadcast algorithm, in the sense that is performed in two steps. Chan et al. [8] proposed an algorithm that takes advantage of architectures with multiple links, where messages can be sent simultaneously over different links in systems with $N$-dimensional meshes/tori. Kandalla et al. [47] also proposed a design for broadcast, reduce and allreduce operations on symmetric scenarios on Xeon Phi. Their design minimizes the use of the PCIe bus and always uses a process on the main processor to communicate with remote nodes. This is the only design specifically targeting Xeon Phi coprocessors. However, it is just valid for hybrid scenarios, involving both Xeon Phi and main processors, and can not be applied to native scenarios (Xeon Phi only), where there are not main processors involved. To solve that Potluri et al. [86] proposed the use of an proxy service running on the main processor, even if the main processor is not used in the job. Intel followed a similar approach with their Coprocessor Communication Link proxy (CCL-proxy) for Intel MPI.

Other works have focused on optimizations for shared memory. Nishtala et al. [77] conducted a series of experiments in three shared memory systems, based on multicore processors, using k-nomial trees for representing the virtual topology of the processes. These experiments demonstrated that for each architecture and message size the optimal radix of the k-nomial trees is different. Graham et al. [33] designed and tested a series of algorithms for shared memory, each one appropriate for a set of functions and message sizes. The described algorithms are basically fan-in or fan-out trees of variable radix; reduce-scatter (each process reduces its data) followed by a gather or all-gather; and a recursive doubling algorithm. Ramos et al. [94] modelled collective communications for cache coherent systems, and proposed enhancements for Xeon Phi taking into account the specific details of its cache implementation. As expected, every algorithm is the best performer for some setups, whereas not optimal for others.

Additionally, there are some works that aimed to optimize both shared and distributed memory architectures, such as the work of Mamidala et al. [65], which implements and evaluates similar algorithms to the previous works. A work closer to the work done on this Thesis is the multi-leader algorithm proposed by Kandalla et al. [45]. This proposal is similar to Bibo Tu's broadcast algorithm, except for using more than one leader per node, initially considering only the allgather operation. Nishtala et al. [78] leveraged shared memory and trees to optimize collectives and explore their autotuning possibilities. Qian [88] followed a similar path to Kandalla et al. and proposed a series of algorithms mainly focused on all-to-all and allgather, targeting multicore systems with multiple connections per node, as well as optimizing the algorithms for cases where different processes arrive at the collective at a different time.

The algorithms considered in the related work are usually independent of their actual implementation in a particular language. However, they have been generally developed using MPI or UPC. Usually there is no algorithm that always outperforms the others. In fact, the performance of an algorithm depends on three factors: (1) message size, (2) number of processes involved, and (3) the hardware, including the network topology . Providing the best algorithm for each setup and message size is the optimum approach, as demonstrated in [106]. However, selecting among the algorithms entails a significant effort, as they are highly dependent on the system. The solution typically relies on autotuning [118], generally based on an automatic performance characterization of each algorithm for a wide range of setups.

Furthermore, it is possible to adapt the runtimes to the underlying hardware. This typically rely on adding software features in order to achieve a better usage of the hardware, or adapting hardware specific layers to a given architecture, e.g., the network layer. Miao et al. [69] proposed a single copy method to take advantage of shared memory architectures, avoiding the system buffer. The proposal of Trahay et al. relies on a multithreaded communication engine to offload small messages [112]. Brightwell et al. [4] propose the sharing of page tables between processes, speeding up applications performance. Hoefler et al. [34] proposed the use of multicast in networks, resulting in highly scalable operations, but just valid for very small messages. Velamati et al. [119] designed a set of algorithms for MPI collective operations for the heterogeneous Cell processor. More recently, Li et al. [53]

have proposed a NUMA-aware multithreaded MPI runtime, where MPI ranks are implemented as threads as opposed to processes, and they have implemented and evaluated algorithms for allreduce in this runtime. Regarding experiences with Xeon Phi and UPC, Luo et al. [57] performed the first general performance evaluation of UPC and MPI on Xeon Phi, slightly covering point to point and collective communication performance, and computational kernels. Potluri et al. [87] optimized MVAPICH2 for Xeon Phi by leveraging the Symmetric Communications InterFace (SCIF), a low level API that allows to control the DMA engines.

The work done up to now successfully adapted collectives to different architectures. However, it does not combine one-sided communications –so common in PGAS languages–, pipelining/overlapping and hierarchical trees. New NUMA architectures and manycore coprocessors with direct access to the network can benefit from these features.

## 1.5. Conclusions of Chapter 1

The scientific community have seen a growing number of programming models in the last years. Some of them address the complexity of process communication in distributed memory systems, incorporating new features. PGAS, with its inherent easiness for remote memory access, is the most promising alternative to traditional message-passing.

Research on collective operations has been an ongoing topic for computer scientist working on optimization of parallel runtimes for a number of years. Typically, changes in the underlying hardware opened a door to new optimizations to better utilize the system resources. Now, with the blooming potential of UPC, a new door has been opened to develop new algorithms that efficiently exploit one-sided remote memory accesses and that are aware of the substantial changes that are taking place in processors architectures.

The next Chapter of this Thesis will characterize the performance of UPC, as a representative PGAS language, to assess its suitability as a language for HPC, and as an alternative to MPI. Most current systems are NUMA clusters, which motivates this characterization to be focused in such architectures.

# Chapter 2

# Characterization of UPC Performance

The PGAS paradigm has been proposed as a programming model with substantial benefits over more traditional models such as message-passing. The distributed shared memory and one-sided memory copies make it an attractive alternative for scientific computing, where programmers typically are more focused on their field of study than on explicitly managing low-level functions. However, without efficient compilers and runtimes that can extract significant performance out of the hardware they are running on, PGAS alternatives such as UPC cannot be considered as a viable approach for future HPC. Nevertheless, the value of the PGAS approach has been already validated. The PGAS extension to Fortran –Co-Array Fortran–, equivalent to UPC for the C language, has been used successfully [72], reaching goals that are not attainable by traditional message-passing and underlining the potential of the PGAS approach. Given the focus on UPC of this Thesis, this Chapter analyzes the behavior of UPC, comparatively with MPI. For this, the most important benchmarks have been used. Section 2.1 explains the benchmarks and analyzes the benchmark results. Section 2.2 exposes additional UPC performance analyses, done by other researchers. Section 2.3 summarizes the conclusions of the Chapter.

## 2.1. Representative Benchmarks for UPC Performance Characterization

The number of benchmarks available that support the direct comparison of UPC and MPI is quite reduced. One of this benchmarks is the de-facto standard for benchmarking HPC systems, the NAS Parallel Benchmarks [2, 73]. Due to its importance and availability they have been used in a wide amount of performance comparisons [1, 43, 64, 93, 95]. Additionally, the NPB implemented in UPC have been already studied in the past, making comparisons between UPC, MPI and OpenMP. Jin et al. [44] developed in UPC their own implementation of the 3 pseudo application codes contained in the NPB –ST, BT and LU–, and compared then with a C implementation of the NPB, using MPI to communicate between processes. They have used the B class –a medium problem size– in 2 different systems. Their findings revealed a performance practically equivalent between the UPC and C+MPI implementations of the NPB. El-Ghazawi and Cantonnet [18] compared the performance of NPB-MPI –implemented in Fortran– with NPB-UPC on a 16 processor Compaq AlphaServer SC cluster, using the class B workload. Cantonnet et al.[6] used two SGI Origin NUMA machines, each one with 32 processors, using the class A workload for 3 NPB kernels. El-Ghazawi et al. [21] compared MPI with UPC very briefly using 2 NPB kernels and class B, in a Cray X1 machine. However, none of these comparisons accounted for modern NUMA hardware using large workloads –class C– as does Subsection 2.1.1. Besides the NPB, matrix multiplication and stencil-like computations are two typical computational kernels widely extended in many scientific applications. Therefore Subsection 2.1.2, assesses the scalability of matrix multiplication kernels in distributed memory paradigms, and a Sobel edge detector.

The testbed used in this analysis is the Finis Terrae supercomputer [107], composed of 142 HP Integrity rx7640 nodes, each one with 8 Montvale Itanium 2 dual-core processors (16 cores per node) at 1.6 GHz and 128 GB of memory, interconnected via InfiniBand. The InfiniBand HCA is a dual 4X IB port (16 Gbps of theoretical effective bandwidth). For the evaluation of the hybrid shared/distributed memory scenario, 8 nodes have been used (up to 128 cores). The number of cores used per node in the performance evaluation is $\lceil n/8 \rceil$, being $n$ the total number of cores used in the execution, with consecutive distribution. An HP Integrity Superdome system with 64 Montvale Itanium 2 dual-core processors (total 128 cores) at 1.6 GHz and 1 TB of memory has also been used for the shared memory evaluation. The nodes were used without other users processes running, and the process affinity was handled by the operating system scheduler.

The MPI library is the recommended by the hardware vendor, HP MPI 2.2.5.1 using InfiniBand Verbs (IBV) for internode communication, and shared memory transfers (HP MPI SHM driver) for intranode communication. The UPC compiler is Berkeley UPC 2.8, which uses the IBV driver for distributed memory communication, and POSIX threads (from now on pthreads) within a node for shared memory transfers. The backend for both and OpenMP compiler is the Intel 11.0.069.

### 2.1.1. NAS Parallel Benchmarks (NPB)

The NPB consist of a set of kernels and pseudo-applications, taken primarily from Computational Fluid Dynamics (CFD) applications. These benchmarks reflect different kinds of computation and communication patterns that are important across a wide range of applications, which makes them the de facto standard in parallel performance benchmarking. There are NPB implementations available for a wide range of parallel programming languages and libraries, such as MPI (from now on NPB-MPI), UPC (from now on NPB-UPC), OpenMP (from now on NPB-OMP), a hybrid MPI+OpenMP implementation (not used in this comparative evaluation as it implements benchmarks not available in NPB-UPC), HPF and Message-Passing Java [64], among others. The most used subset of the NPB are the kernels Conjugate Gradient (CG), Embarrassingly Parallel (EP), Fourier Transform (FT), Integer Sort (IS) and Multi Grid (MG). All these kernels use double precision, except IS, that operates with integer data. Additionally, the original NPB specification also contained three pseudo-applications: Block Tridiagonal solver (BT), Scalar Pentadiagonal solver (SP) and Lower-Upper Gauss-Seidel solver (LU). For all these benchmarks there are different sizes defined, ranging from class A –the smaller– to class E.

The NPB selected for evaluation are: CG, EP, FT, IS and MG. The CG kernel is an iterative solver that tests regular communications in sparse matrix-vector multiplications. The EP kernel is an embarrassingly parallel code that assesses the floating point performance, without significant communication. The FT kernel performs series of 1-D FFTs on a 3-D mesh that tests aggregated communication performance. The IS kernel is a large integer sort that evaluates both integer computation performance and the aggregated communication throughput. MG is a simplified multigrid kernel that performs both short and long distance communications. Moreover, each kernel has several workloads to scale from small systems to supercomputers. NPB-MPI and NPB-OMP are implemented using Fortran, except for IS which is programmed in C. The fact that the NPB are programmed in Fortran has been considered as cause of a poorer performance of NPB-UPC [18], due to better backend compiler optimizations for Fortran than for C.

Most of the NPB-UPC kernels [27] have been manually optimized through techniques that mature UPC compilers should handle in the future: privatization, which casts local shared accesses to private memory accesses, avoiding the translation from global shared address to actual address in local memory, and prefetching, which copies non-local shared memory blocks into private memory.

### Performance of NPB Kernels on Hybrid Memory

Figure 2.1 shows NPB-MPI and NPB-UPC performance on the hybrid configuration, using both InfiniBand and shared memory communication in the Finis Terrae supercomputer. The left graphs show the kernels performance in MOPS (Million Operations Per Second), whereas the right graphs present their associated speedups.

Regarding the CG kernel, MPI performs slightly worse than UPC using up to 32 cores, due to the kernel implementation, whereas on 64, and especially on 128 cores MPI outperforms UPC. Although UPC uses pthreads within a node, its communication operations, most of them point-to-point transfers with a regular communication pattern, are less scalable than MPI primitives, due to the contention caused by many pthreads trying to communicate using the same instance of the UPC runtime.

EP is an embarrassingly parallel kernel, and therefore shows almost linear scalability for both MPI and UPC. The results in MOPS are approximately 6 times lower for UPC than for MPI due to the poorer UPC compiler optimizations. EP is the only NPB-UPC kernel that has not been optimized through prefetching and/or pri-

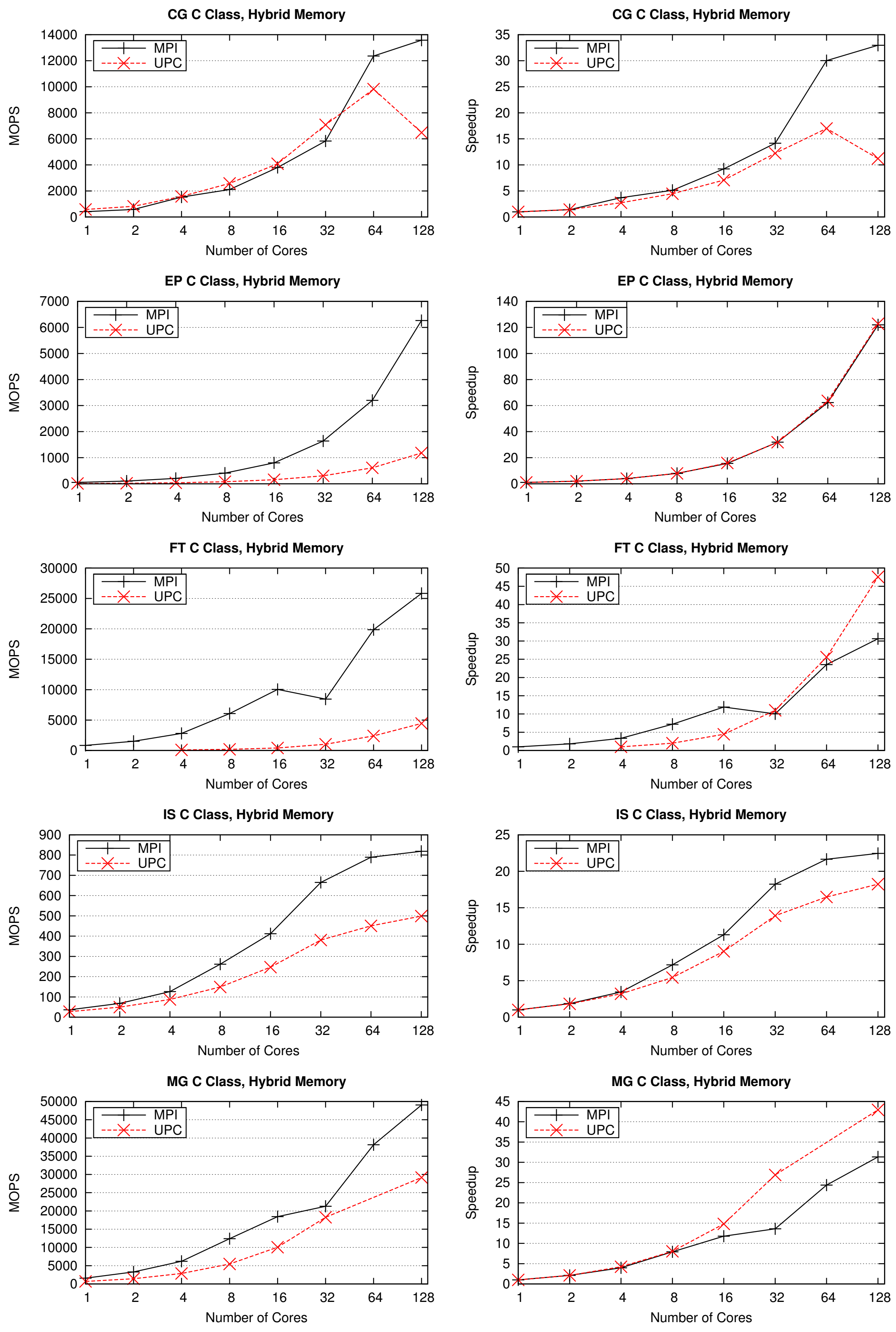


Figure 2.1: Performance of NPB kernels on hybrid shared/distributed memory

vatization, and the workload distribution is done through a `upc_forall` construct, preventing more aggressive optimizations. Besides this, most of the time the EP kernel is generating random numbers, meaning that it is compute bound. In this case, the backend language (Fortran as opposed to C), can play a major role, as demonstrated in [12, 44].

The performance of FT depends on the efficiency of the exchange collective operations. Although the UPC implementation is optimized through privatization, it presents significantly lower performance than MPI. The UPC results, although significantly lower than MPI in terms of MOPS, show higher speedups than MPI. This is a communication-intensive code that benefits from UPC intranode shared memory communication, which is maximized on 64 and 128 cores.

The IS kernel is a quite communication-intensive code. Thus, both MPI and UPC obtain low speedups for this kernel (less than 25x on 128 cores). Although UPC IS has been optimized using privatization, the lower performance of its communications limits its scalability, which is slightly lower than MPI speedups.

Regarding MG, MPI outperforms UPC in terms of MOPS, whereas UPC shows higher speedup. The reason is the poor performance of UPC MG on one core, which allows it to obtain almost linear speedups on up to 16 cores.

### Performance of NPB Kernels on Shared Memory

Figure 2.2 shows NPB performance on the Superdome system. As in the hybrid memory figures, the left graphs show the kernels performance in MOPS and the right graphs show the speedups. MPI requires copying data on shared memory, and therefore could be considered less efficient than the direct access to shared memory of UPC and OpenMP. The following results do not support this hypothesis.

Regarding CG, all options show similar performance using up to 32 cores. However, for 64 and 128 cores UPC scalability is poor, whereas MPI achieves the best MOPS results. The poor performance of OpenMP on one core leads OpenMP to present the highest speedups on up to 64 cores, being outperformed by MPI on 128 cores.

As EP is an embarrassingly parallel code, the scalability shown is almost linear for

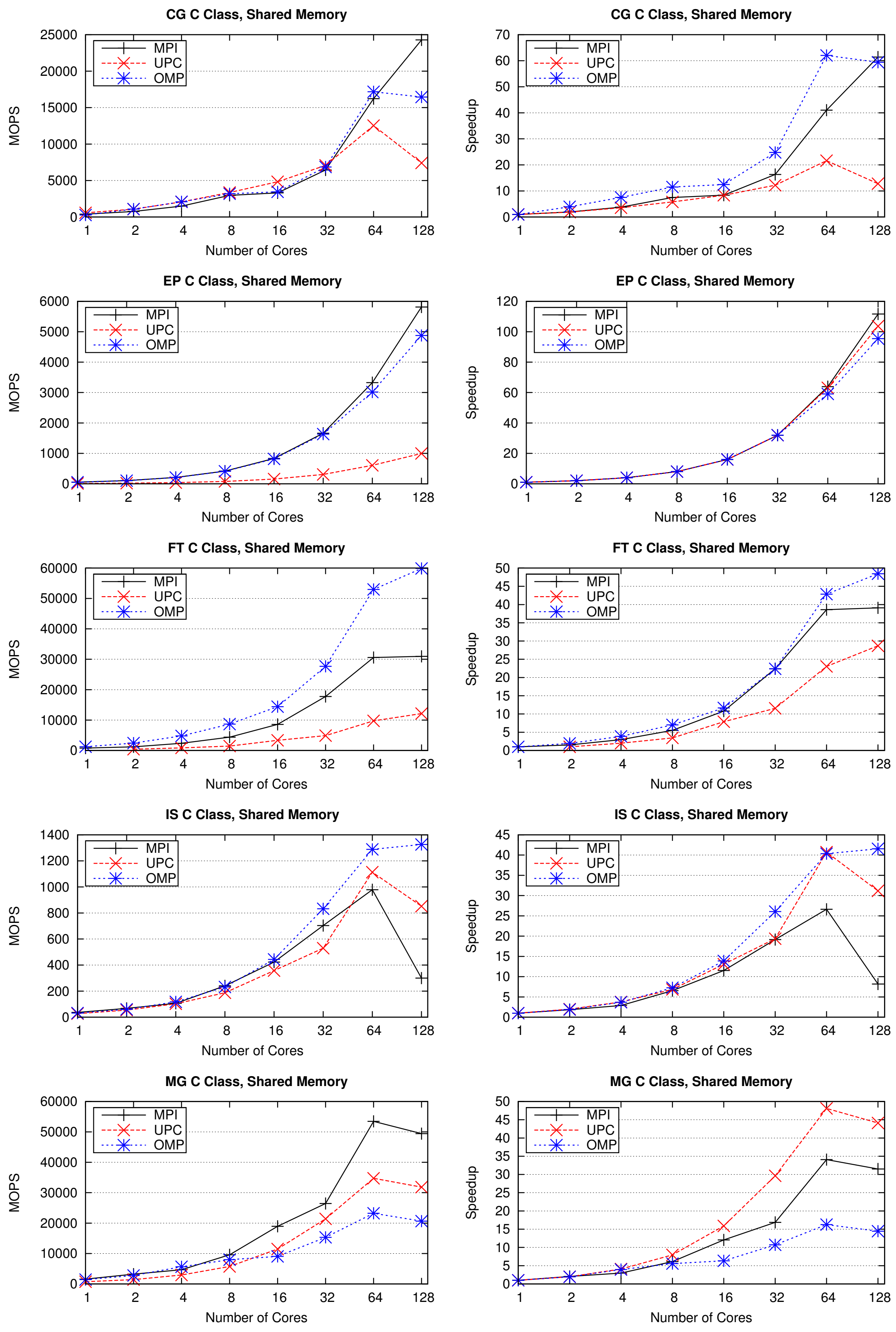


Figure 2.2: Performance of NPB kernels on shared memory

MPI, UPC, and OpenMP, although MPI obtains slightly higher speedups, whereas OpenMP presents the lowest scalability. These results are explained by the efficiency in data locality exploitation of these three options. In terms of MOPS, UPC shows quite low performance, as already discussed in subsection 2.1.1.

As FT is a communication-intensive code, its scalability depends on the performance of the communication operations. Therefore, OpenMP and MPI achieve high speedups, whereas UPC suffers from a less scalable exchange operation. The code structure of the OpenMP implementation allows more efficient optimizations and higher performance. Due to its good scalability OpenMP doubles MPI performance (in terms of MOPS) on 128 cores. UPC obtains the poorest performance.

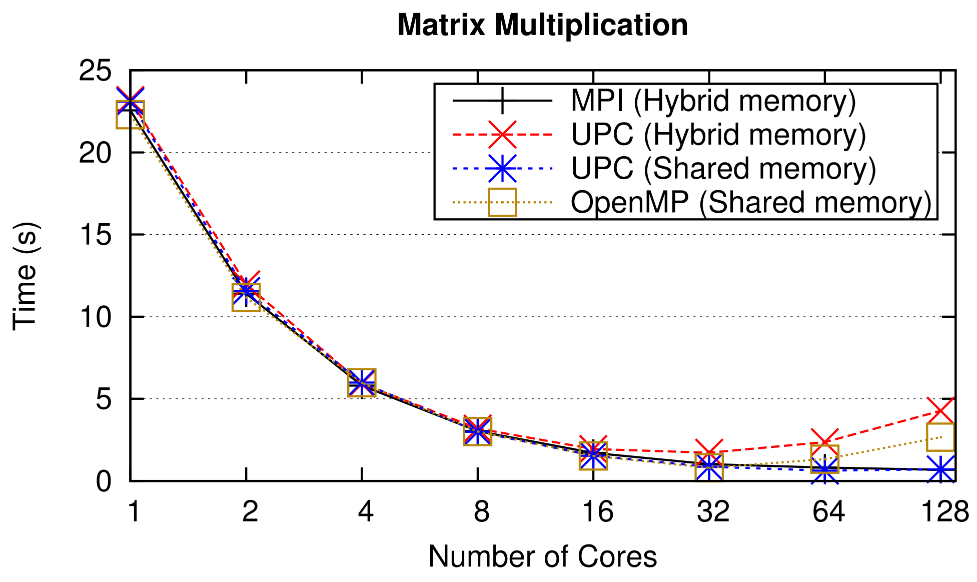
IS is a communication-intensive code that shows similar performance for MPI, UPC and OpenMP on up to 32 cores, both in terms of MOPS and speedups, as the results on one core are quite similar among them. This fact can be partly explained by the fact that the IS kernels use the same backend compiler (icc). Regarding 64 and 128 cores results, OpenMP obtains the best performance and MPI the lowest, as the communications are the performance bottleneck of this kernel.

Regarding MG, MPI achieves better performance in terms of MOPS than UPC and OpenMP, whereas UPC obtains the highest speedups, due to the poor performance of this kernel on one core. OpenMP shows the lowest results, both in terms of MOPS and speedups.

### 2.1.2. Matrix Multiplication and Stencil Kernels Benchmarks

Matrix multiplications and Stencil kernels are widely extended in scientific computation. Therefore, versions of these kernels were developed for UPC and MPI [25]. The matrix multiplication kernel implements a simple matrix multiplication algorithm with blocking. The Stencil kernel implements the Sobel edge detector algorithm, widely used in image processing, and that computes the gradient of the image intensity function, relying just in integer data types.

Figures 2.3 and 2.4 show the results for the matrix multiplication and the Sobel kernel. The matrix multiplication uses matrices of 2400×2400 doubles, with a blocking factor of 100 elements, and the experimental results include the data dis-



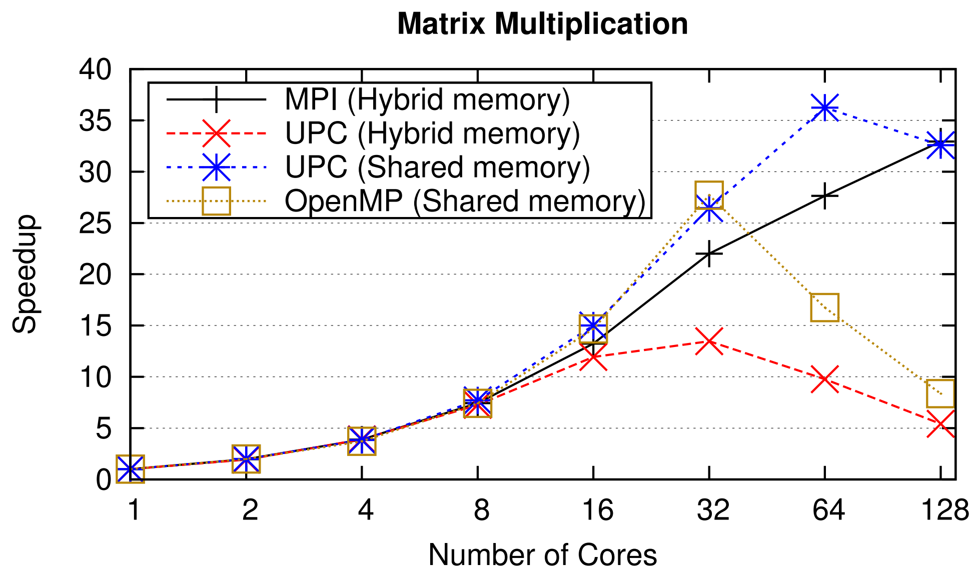
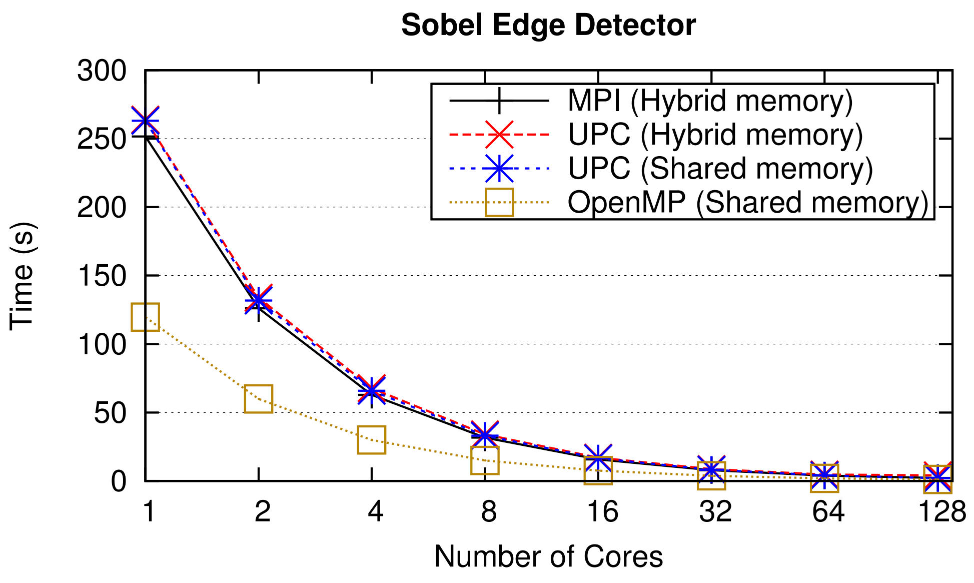


Figure 2.3: Performance of UPC, MPI and OpenMP matrix multiplication implementations in Finis Terrae



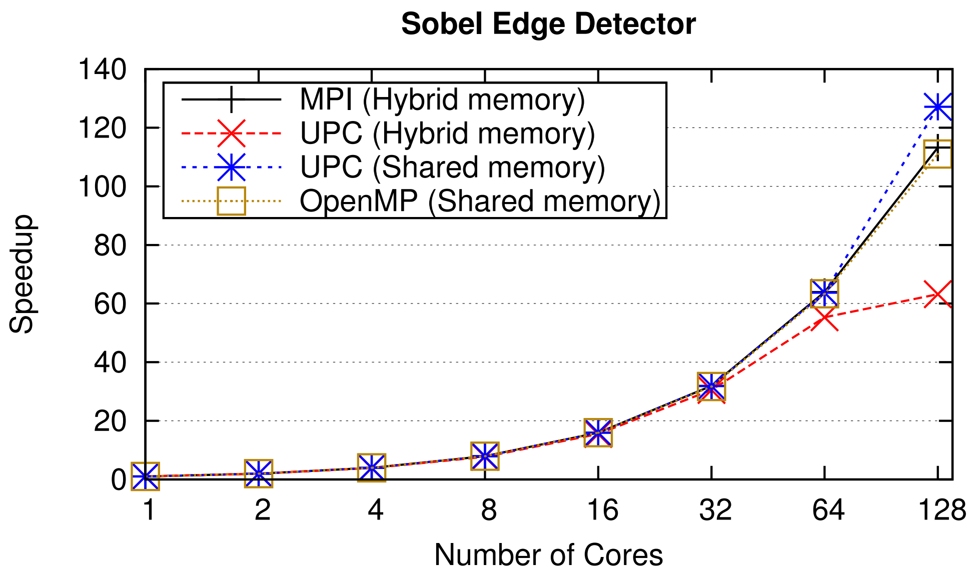


Figure 2.4: Performance of UPC, MPI and OpenMP Sobel kernels implementations in Finis Terrae

tribution overhead. The Sobel kernel uses a 65536×65536 unsigned char matrix and does not take into account the data distribution overhead. The graphs show the speedups on the hybrid scenario (MPI and UPC) and in the shared memory system (UPC and OpenMP). The plots at the left show run time, whereas the plots at the right show scalability.

The three programming models (PGAS, MPI and OpenMP) obtain similar speedups on up to 8 cores. MPI can take advantage of the use of up to 128 cores, whereas UPC (hybrid memory) presents poor scalability. In shared memory, UPC and OpenMP show similar speedups up to 32 cores. However, on 128 cores UPC achieves the best performance, whereas OpenMP suffers an important performance penalty due to the sharing of one of the matrices. However, in UPC, this matrix is copied to private space, thus avoiding shared memory access contention. MPI shows better performance than OpenMP for this reason.

In the Sobel kernel results, because the data distribution overhead is not considered, the speedups are almost linear, except for UPC on the hybrid scenario, where several remote shared memory accesses limit seriously its scalability. Nevertheless, UPC on shared memory achieves the highest speedups as these remote accesses are intraprocess accesses (UPC uses pthreads in this scenario).

## 2.2. Other UPC Performance Studies

During the last years the number of published work about UPC performance has increased. Different studies have been made available, showing the performance of UPC on diverse mathematical operations and libraries. González et al. have studied the performance of their UPC mathematical libraries. This way, in [29] they have evaluated the performance of their Cholesky and LU solvers against the equivalent in ScaLAPACK, concluding that the performance of their implementation when compared to a similar ScaLAPACK implementation –relying in 1D decomposition– is better, whereas for 2D, even though ScaLAPACK performs better, the performance of the UPC solution is a good compromise between performance and programmability. In [30] González et al. evaluated the performance of triangular solvers, reaching similar conclusions.

Jansson has developed JANPACK [39], an sparse matrix library developed in UPC, and his studies have concluded that JANPACK is typically twice as fast as PETSc for two different set of benchmarks.

Teijeiro et al. developed a MapReduce framework in UPC [103], and evaluated it in shared and distributed memory systems, using four representative applications typically used in MapReduce environments. Their conclusion is that their framework achieved similar performance to leading MapReduce implementations, sometimes obtaining even better performance than them. Teijeiro et al. also implemented a Brownian dynamics application in UPC. Their evaluation of this application [101, 102], comparatively with the same application implemented in OpenMP and MPI, have shown that UPC scales further than OpenMP, and its performance rivals with MPI.

## 2.3. Conclusions of Chapter 2

The performance evaluation presented in this Chapter has shown that even though MPI typically performs better, UPC can achieve competitive performance and scalability, depending on the particular computational and communication workload. Additionally, the performance differences cannot be completely justified by the programming model. The EP (Embarrassingly Parallel) kernel for instance, is not a communication intensive code, yet the MPI version performs better, which can be just explained by single thread performance, meaning that the effectiveness of the underlying compiler and language (C vs. Fortran) play an important role in the performance and low level optimization. Similar experiments comparing another PGAS implementation of the NPB [12], using Co-array Fortran, and comparisons of C+MPI and UPC implementations of the NPB [44] supports this statement.

Moreover, recent works have shown promising performance in many contexts. Even though message passing is likely to remain as the most widely used paradigm for HPC, PGAS, and in particular UPC, has proved itself as a viable alternative, particularly due to the possibilities that its integrated one-sided communications and remote memory access bring.

In order to be widely accepted, the implementations of the PGAS model have to provide efficient collective operations. However, without specific ways to assess the performance and improvement of different collective algorithms, the research on this field lacks an important tool. The next Chapter presents the UPC Operations MIcrobenchmarking Suite (UOMS), a tool developed as a prerequisite to evaluate collective algorithms.

# Chapter 3

# UPC Operations Microbenchmarking Suite: UOMS

The UPC community lacks so far of tools for assessing the performance of UPC operations. There are tools for validating some APIs [24], implementations of the NPB kernels [27] –a suite of problems widely extended in HPC, used for comparisons across systems and languages and used in the previous Chapter– and other sets of benchmarks including matrix multiplication, Sobel edge detector, N-queens [25], as well as an UPC version of the Scalable Synthetic Compact Application (SSCA) benchmarks [26]. However, none of this tools allow to evaluate the performance of discrete communication functions. Microbenchmarking is an important tool to characterize in an isolated way the performance of different parts of a runtime or system. Therefore, before developing new collective algorithms, it is mandatory to have a microbenchmarking tool that allows to measure the performance of these algorithms in a methodical and comparative way. This Chapter presents the design of UPC Operations Microbenchmarking Suite (from now on UOMS), a suite developed to cover this gap. A state-of-the-art section is next presented, where the most popular microbenchmarking tools are presented and categorized. Section 3.2 provides then an overview of the design of UOMS. Section 3.3 explains the different benchmarks supported by UOMS. Section 3.4 shows the different tuning parameters available in UOMS. Finally, Section 3.5 concludes the Chapter.

## 3.1. Benchmarking Suites in HPC

Benchmarking of HPC systems is typically a complex task. All the different components, both hardware and software, frequently interact with each other, and understanding the performance characteristics of a given system is far from trivial. Moreover, comparison of different supercomputers and architectures has to be done in a fair manner, in order to obtain meaningful results.

To have a bottom-up comprehensive understanding of performance of high performance computers, benchmarks that focus on particular aspects are frequently used. The most commonly benchmarked parts of a supercomputer are:

- Processor/floating point units
- Memory subsystem
- Communications
- Input and output (to permanent storage)

To assess the processor performance DGEMM, a matrix-matrix multiplication function of the BLAS level 3 library [16] is frequently used for its importance in many HPC applications and its simplicity. Likewise, the FFT benchmark [66] is used to assess the performance of FFTW implementations. The High Performance LINPACK benchmark [15] is the most popular benchmark for HPC, in fact it is used to rank supercomputers in the top 500 list [111], and focuses almost exclusively on processor performance. It solves a dense system of linear equations, and therefore its communication/computation ratio is close to 0.

The STREAM benchmark developed by McCalpin [67] is a popular benchmark used to measure sustained memory bandwidth, accessing vectors in 4 different ways: (1) copying vectors `a[i] = b[i]`, (2) scaling vectors `a[i] = q*b[i]`, (3) adding vectors `a[i] = b[i] + c[i]`, and (4) adding a vector and a scaled vector `a[i] = b[i] + q*c[i]`. Its popularity resides in its simplicity, that allows to analyze easily the behavior of the difference cache levels and main memory. It has been implemented in a number of languages and programming models, being the most populars the OpenMP and the MPI implementations. More recently Jalby et. al presented WBTK [38], that allows for benchmarking on a wider parameter space, including different memory strides. The RandomAccess benchmark by Koester and Lucas [14] covers some of the space not covered by other tools, and assess the performance of random memory access.

Regarding communications for HPC the reference are the Intel MPI Benchmarks [37] (from now on IMB), previously known as Pallas MPI benchmarks. This benchmarks test a range of MPI functions, including point to point, collectives and barriers, and has been widely used as the de-facto standard for MPI communications for a long time. The Ohio State University (OSU) Microbenchmarks [54, 81] are a set of benchmarks also focused on the performance of MPI functions, that includes tests on multithreaded environments, CUDA/OpenACC environments, and that has extended its scope to cover some basic UPC and OpenSHMEM functions recently. Bull et. al also implemented their own multithreaded MPI microbenchmark [5].

Performance of I/O can be also tested with the Intel MPI benchmarks, when using the MPI-IO interface. The Effective I/O Bandwidth Benchmark by Rabenseifner and Koniges [91] is also used to measure MPI-IO performance, in a more dedicated and extended way. The IOR benchmark [52] is a more versatile and used benchmark for parallel I/O, that supports different interfaces, such as POSIX, MPI-IO, HDF5 and PnetCDF.

The number and variety of benchmarking tools shows the importance of benchmarking and microbenchmarking on HPC. However, the UPC community, due to its relative short age, did not have a comprehensive and methodical benchmark that allows to measure systematically and reliably its communications performance. This lack of benchmarking tools is the main motivation of the development of the UPC Operations Microbenchmarking Suite (from now on UOMS), as there were no other UPC communications benchmarks available before its development.

## 3.2. Design of UPC Operations Microbenchmarking Suite

During the design of UOMS it was decided to follow a similar approach as IMB. There are three reasons for this decision: (1) to make easier to understand the data reported by the benchmark for users with previous knowledge of IMB, (2) because IMB provides a comprehensible and complete set of results for a wide variety of scenarios, and (3) to allow direct comparisons with leading MPI implementations. This way, UOMS reports latency and bandwidth for different functions, using a range of message sizes.

However, IMB and UOMS have small differences in how they measure performance. IMB reports minimum, maximum and average latency. However, this data is the average per message size per process. The formulas of the reported data are described in Equation 3.1, where $p$ is the number of processes and $n$ is the number of iterations for a given message size. UOMS also reports minimum, maximum and average latencies. However, these latencies are considering iterations, not processes, as UOMS by default considers one operation finished just when all the processes involved are done, using `UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC` as synchronization mode. The formulas for the reported data in UOMS are described in Equation 3.2. In order to allow comparisons as fair as possible the values used for IMB should the maximum, i.e. the highest average time among processes, to guarantee a state where all the processes have finished the operation. The comparable values for UOMS are the average, i.e. the average time per iteration needed to guarantee that all the processes have finished the operation. Both reflect the average time needed to allow the operation to be completed by all the processes.

$$\min_{i=1}^{p}\left(\frac{\sum_{j=1}^{n} l_j}{n}\right)_i \qquad \max_{i=1}^{p}\left(\frac{\sum_{j=1}^{n} l_j}{n}\right)_i \qquad \frac{\sum_{i=1}^{p}\left(\frac{\sum_{j=1}^{n} l_j}{n}\right)_i}{p} \tag{3.1}$$

$$\min_{i=1}^{n}\left(\max_{j=1}^{p} l_j\right)_i \qquad \max_{i=1}^{n}\left(\max_{j=1}^{p} l_j\right)_i \qquad \frac{\sum_{i=1}^{n}\left(\max_{j=1}^{p} l_j\right)_i}{n} \tag{3.2}$$

The reasons why UOMS, by design, differs from IMB are three: (1) the users might missinterprete the metrics, as intuitively the minimum, maximum and average are with respect to the number of iterations, not the individual process results, (2) the most common scenario uses `UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC` as synchronization mode, and therefore the time will be very similar for all the threads, and (3) when using UOMS to understand collective performance impact within an application reporting data with respect to individual processes performance is troublesome, as load imbalance between processes (very different time for minimum and maximum time with respect to processes) will overcomplicate the analysis, contrarily to the nature of a microbenchmark.

Another difference that requires attention is that the root of each collective in IMB changes every iteration, whereas UOMS keeps the root static. The vast majority of collectives use rank/thread 0 as root, and therefore, UOMS does not round robin the root on each iteration.

UOMS aims to be a reliable and easy to use suite of microbenchmarking. It is designed to be easily extended, modified and maintained. To allow its users to accomplish this target, it has been released under the GNU General Public License (GPL).

## 3.3. UOMS Benchmarking Units

UOMS consist of different benchmarking "units", which, according to their nature can be grouped in four different sets:

**Memory operations** : This unit tests `upc_all_alloc` and `upc_free` functions. Memory allocation and freeing is an expensive operation in every system, especially in PGAS, due to the internal synchronization across processes/threads required to maintain the global view of the memory. Some algorithms might require efficient allocation and freeing schemes for temporal buffering, in order to scale and have a good performance. Therefore, benchmarking these operations is an important task. The results reported are the overhead of allocating/freeing chunks of memory varying its size.

**Discrete accesses** : This unit tests the latency to access discrete elements in a shared array. The tests include, read, write and read+write operations. Besides this, two different types of tests are performed: `upc_forall` and `for`. The first one distributes the workload across all the UPC threads. In the second one the whole operation is performed by thread 0. This is useful for testing the speed of remote accesses and optimization techniques such as coalescing. The benchmarking of the read operation consists of a sum of a scalar variable in the stack and the elements of an array, to prevent the compiler from dropping the first $N-1$ iterations. The operation performed in write is a straight forward update of the elements of an array. The operation performed in read+write is a sum of the current element and its position in the array. The data type used is `int`.

**Block accesses** : This unit tests the performance of memory movement operations, namely `upc_memput`, `upc_memget` and `upc_memcpy`. There are variations of these tests. For each one of them, two tests are done: remote and local access. In this case, when two threads are used, affinity tests are also performed. This way the effects of data locality in NUMA systems can be measured, whenever the two threads run in the same machine, although this feature may be useful even if the two threads run in different machines. E.g.: Machines with non-uniform access to the network interface, like quad-socket Opteron/Nehalem-based machines, or cell-based machines like HP Integrity servers. The non-blocking variants –both with explicit and implicit handler– proposed by Berkeley UPC and included in the UPC specification version 1.3 are also tested. In order to provide a framework of reference for the overhead introduced by the runtime, the performance of the system calls `memcpy` and `memmove` is also examined.

**Collective operations** : This unit is used to measure the performance of all the collective operations defined in the UPC specification –broadcast, scatter, gather, gather all, permute, exchange, reduce and prefix reduce– including all the different variants for reduce and prefix reduce operations. I.e. for the following data types: `char`, `unsigned char`, `short`, `unsigned short`, `integer`, `unsigned integer`, `long`, `unsigned long`, `float`, `double` and `long double`. Additionally, the `upc_barrier` function is also tested.

Figure 3.1 illustrates the output produced by UOMS, using the scatter function. The header indicates the benchmarked function and the number of processes involved. The first column shows the message size (block size) used for each particular row. The second column is the number of repetitions performed for that particular message size. The following three columns are, respectively, the minimum, maximum and average latencies. The last column shows the aggregated bandwidth calculated using the minimum latencies. Therefore, the bandwidth reported is the maximum bandwidth achieved in all the repetitions.

In point to point block memory copies the output shows the affinity mask (pinning) of the communicating threads:

```
#---------------------------------------------------------
# using #cores = 0 and 1 (Number of cores per node: 16)
# CPU Mask: 1000000000000000 (core 0), 0100000000000000 (core 1)
#---------------------------------------------------------
```

This indicates that all the tests after these lines are performed using core 0 (thread 0) and core 1 (thread 1) until another affinity header is showed.

The output depicted in Figure 3.1 is common for most of the benchmarks. However, the output for the barrier benchmark does not contain the bandwidth and the message size columns, since it is a dataless operation. Similarly, reduce and prefix reduce benchmarks, contain the message size column, but not the bandwidth column, as it is highly dependent on the algorithm and is not a reliable metric for these functions.

Most of the operations show the bandwidth achieved, which is calculated taking into account the particular function tested, as each data movement is different. As a general rule, bandwidth is calculated as $(factor * message\ size)\ /time$. Using this formula, $factor$ equals to $THREADS$ for most of the functions. For exchange and gather all $factor$ is $THREADS * THREADS$, as they are all to all collectives. For point to point functions $factor$ is 1.

With the output produced by UOMS, the UPC research community has a powerful and comprehensible tool to analyze the communications performance of any given UPC runtime. UOMS supports correlation of data, and analysis of the impact

```
#---------------------------------------------------
#    UPC Operations Microbenchmark Suite V1.1
#---------------------------------------------------
# Date                     : Sun Oct  6 18:09:31 2013
# Machine                  : x86_64
# System                   : Linux
# Release                  : 2.6.32-358.el6.x86_64

# Cache invalidation       : Disabled

# Warmup iteration         : Enabled

# Problem sizes:
# 4
# 8
# 16
# 32
# 64
# 128
# 256
# 512
# 1024
# 2048
# 4096

# Synchronization mode          :   UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC

# Reduce Op                     :   UPC_ADD

# List of Benchmarks to run:

# upc_all_scatter

#-------------------------------------------------
# Benchmarking upc_all_scatter
# #processes = 4096
#-------------------------------------------------
       #bytes  #repetitions     t_min[nsec]     t_max[nsec]     t_avg[nsec]     Bw_aggregated[MB/sec]
            4            40        19870000        33914000     23665850.00                      0.82
            8            40        19870000        39533000     22379525.00                      1.65
           16            40        20171000        32459000     23616650.00                      3.25
           32            40        20171000        33051000     23285975.00                      6.50
           64            40        20460000        43224000     23833875.00                     12.81
          128            40        23283000        37407000     26042950.00                     22.52
          256            40        23578000        34185000     26179325.00                     44.47
          512            40        23951000        43695000     26645500.00                     87.56
         1024            40        25427000        39755000     27943625.00                    164.95
         2048            40        26450000        35300000     28769900.00                    317.15
         4096            40        27395000        37900000     29671125.00                    612.42
```

Figure 3.1: Output example of UOMS

of the usage of multiple threads and the message size on the performance of a particular operation. This way, plots such as the one in Figure 3.2 can be generated, to analyze functions performance.

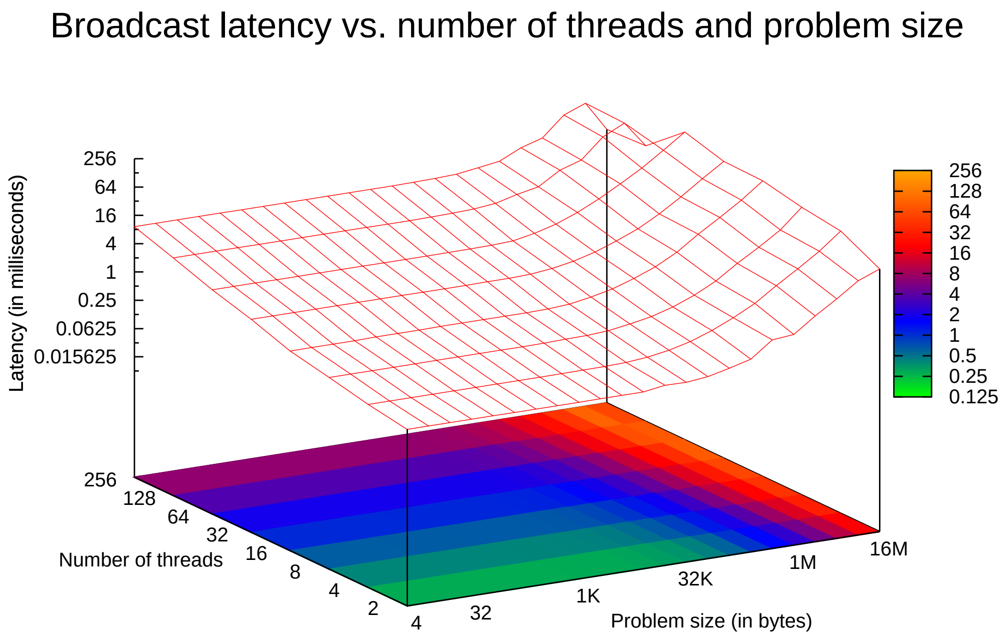


Figure 3.2: UPC broadcast latency on Finis Terrae (3D plot example)

## 3.4. UOMS Options and Compiling Parameters

UOMS aims to be a portable suite, and allows its users to tailor their tests to their needs. Thus, some of the UOMS features can be adjusted at compile time. The number of cores in the system is typically detected using the `_SC_NPROCESSORS_ONLN` option of the `sysconf` system call. However, since this particular option of `sysconf` might not be available in every platform, this can be overridden at compilation time.

The UPC specification version 1.2 did not include non-blocking point to point memory transfers. As a result, many implementations without full support for the 1.3 specification do not include these non-blocking operations. Therefore, support for them in UOMS is optional.

The default message size limits used by most benchmarks in UOMS are 4 bytes (smallest message size limit) and 16 MB (largest message size limit). However, the default limits can be easily changed at compile time.

Besides the compile-time options, UOMS also provides run-time options. This way, UOMS users can specify if they want to communicate cached or uncached data. Likewise, it is possible to include warm up iteration when initializing the benchmark execution. This is useful for letting the first call to a library initialize all its data structures without incurring into penalty in the measurements.

The reduce and prefix reduce operations can use different operators. The default is `UPC_ADD`. However, `UPC_MULT`, `UPC_LOGAND`, `UPC_LOGOR`, `UPC_AND`, `UPC_OR`, `UPC_XOR`, `UPC_MIN` and `UPC_MAX` are operators that can be chosen at run-time.

In a similar fashion the synchronization mode for collective operations is also user selectable. The default is to use a complete synchronization at both the beginning and the end of the operation (`UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC`). At run-time more loose synchronization options are also available, allowing to specify `UPC_{IN|OUT}_MYSYNC` options, where the collective might start to read or write data just from threads that had already entered the collective, or return if no data from the returning thread will be read or written again by the collective. Likewise, the `UPC_{IN|OUT}_NOSYNC` options, that do not impose any synchronization restriction, are also selectable.

The minimum and maximum message sizes set at compile-time can be overridden at run-time. UOMS will calculate the intermediate range using increments of a power of 2. In certain situations the UOMS user might want to specify particular message sizes to be tested. This is possible providing an input file with the desired message sizes. This overrides the automatic message sizes calculated using the minimum and maximum limits given at compile-time or run-time.

The last options provided by UOMS allow users to specify which particular benchmarks they want to run, using an input file, and the maximum amount of time spent per message size.

## 3.5. Conclusions of Chapter 3

UOMS covers an important gap in the benchmarking landscape for UPC. It is the first microbenchmarking suite for UPC, covering a wide range of options, from point to point to collectives, including NUMA and node awareness, work distribution with `upc_for_all` and plain shared memory access in read, write and read+write modes. With UOMS –available at [60]– research on run time and collective algorithm improvements in UPC has now an essential tool.

Recently others microbenchmarking tools are including UPC operations in their tests. The OSU benchmarking suite have included UPC memput and memget operations in their suite in their version 3.8. Recently (24/03/2014), according to the OSU changelog [82], they have also included a subset of the collectives, even though in the source they are not present. Despite the fact that the OSU benchmarks have started to look into the direction of UPC microbenchmarking, UOMS is a more extensive, complete and detailed suite, and aims therefore to be the de facto runtime and collective benchmark suite for UPC.

Next Chapter presents the design and development of a set of PGAS collective algorithms, using different optimization techniques. This performance optimization has relied on UOMS to assess its effectiveness.

# Chapter 4

# Design of Scalable PGAS Collective Algorithms

Current popular UPC collective libraries have limited scalability and performance. This becomes evident when comparing with MPI collectives in NUMA clusters. In [100] an early prototype of UOMS was used to evaluate the performance of the Berkeley UPC collectives [51] and the Michigan Technological University reference implementation [70]. This way, it has been found that in this environment UPC collectives are always significantly outperformed by MPI collectives, using just 32 cores distributed between 4 nodes. This lack of performance motivates the design of new algorithms, able to perform and scale better than the existing implementations.

The design of scalable algorithms depends on two prerequisites: (1) Deep understanding of the operation being optimized, and (2) deep understanding of the underlying hardware characteristics. Additionally, a desirable design principle is the portability of the algorithm, the availability of running on different hardware. Therefore, the design and development of the proposed collectives has been done based on standard UPC constructs. In this Chapter, Section 4.1 provides an overview of the functionality covered by the different collectives present in the UPC standard. Section 4.2 describes the design of the proposed collective algorithms. Finally, Section 4.3 summarizes the conclusions.

## 4.1. UPC Collective Operations

The UPC specification version 1.2 considers the collectives library an essential (non-optional) part of every UPC implementation in order to conform with the UPC specification. Previously it was not included in the specification, but rather as an extra library. This promotion of the collective library is due to its central importance for the application programmer. The collectives in UPC are divided in two groups: relocalization operations and computational operations. The collectives are described in the following subsections.

### 4.1.1. Relocalization Operations

The most common relocalization operation is broadcast. It has the following interface: `void upc_all_broadcast(shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);`. The broadcast function copies a block of `nbytes` of a shared array with affinity to any thread to the corresponding block in a destination shared array, for every thread. Its functionality is sketched in Figure 4.1. At the end of this operation every thread have the same data, and the source is a single thread.

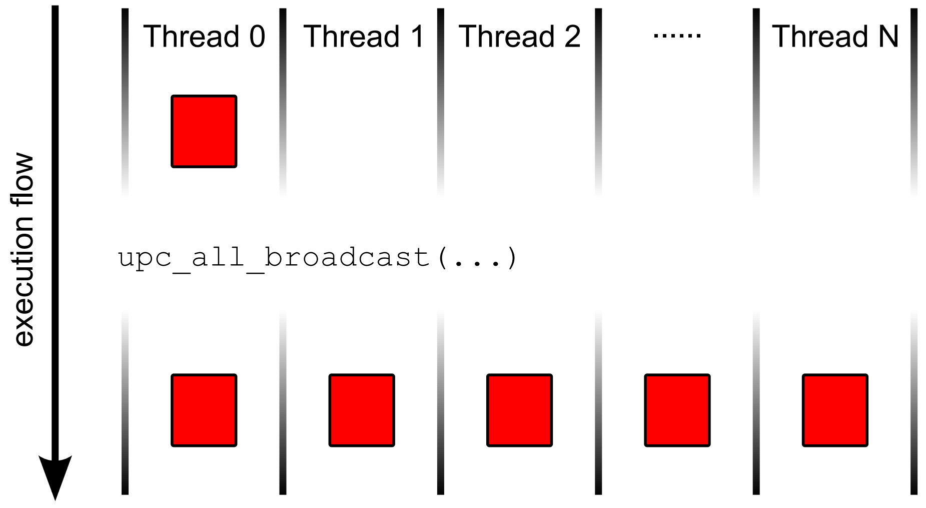


Figure 4.1: Broadcast operation in UPC

Scatter is another relocalization operation. The interface of scatter is as follows:

`void upc_all_scatter(shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);`. Scatter slices an array with affinity to a single thread into blocks of size `nbytes`. Afterwards, this blocks are distributed in a round robin fashion between all threads. At the end of the operation every thread has an unique piece of data, copied from a single thread, as observed in Figure 4.2. This means that typically the source arrays can not be too large in cases with a high number of threads.

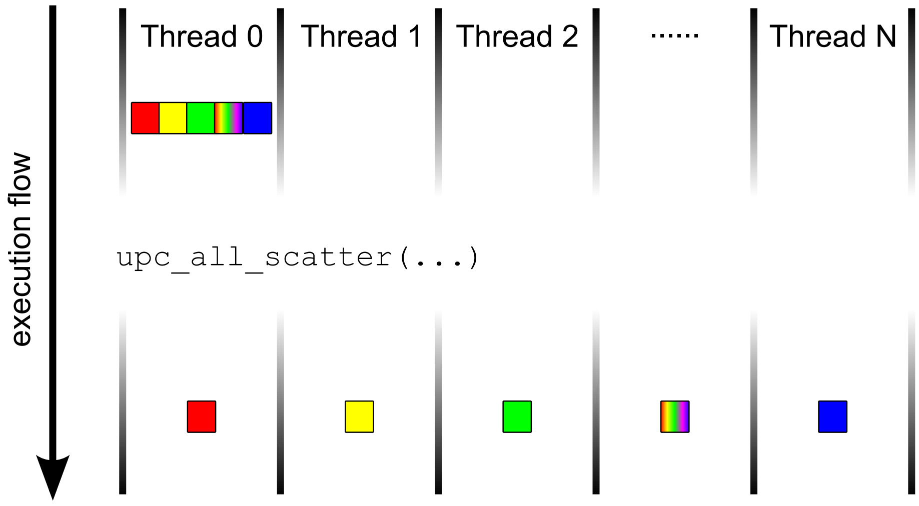


Figure 4.2: Scatter operation in UPC

Gather is the inverse function of scatter. The interface is: `void upc_all_gather (shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);`. In gather, blocks of size `nbytes` and distributed among every thread are collected in an array with affinity to a single thread, in an orderly fashion. When the operation is done a single thread has the aggregated data, while the others keep their own piece. This is depicted in Figure 4.3. As in scatter, this means that the amount of data contributed by each thread can not be too large, as they would not fit into the memory of the destination thread.

Gather all has the following interface: `void upc_all_gather_all(shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);`. Gather all is the functional equivalent of a gather followed by a broadcast, as in the end of the operation every thread has the same data, that is, the result of gathering their individual data in an orderly fashion, as can be observed in Figure 4.4.

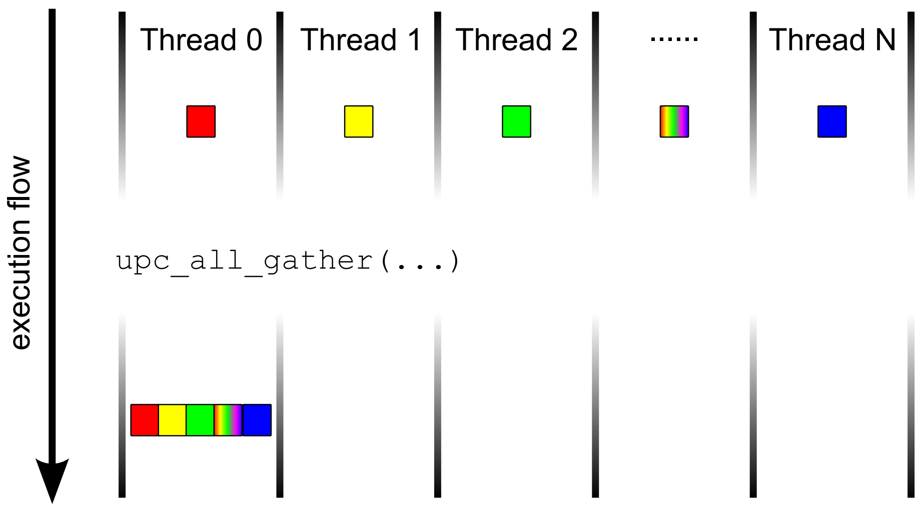


Figure 4.3: Gather operation in UPC

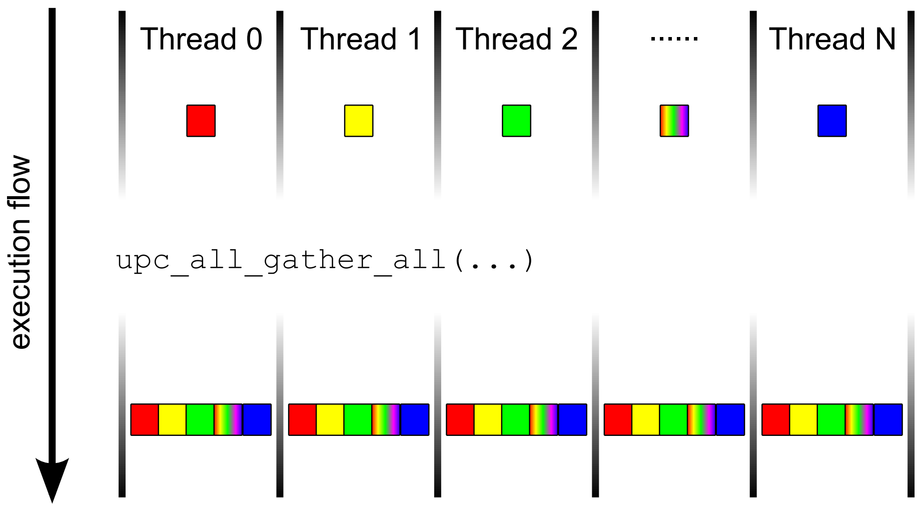


Figure 4.4: Gather all operation in UPC

Exchange is a relocalization operation with this interface: `void upc_all_exchange (shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);`. The motivation behind the exchange operation is to compute a transpose, similarly to what an `alltoall` function computes in MPI. This way, the $i^{th}$ slice of the array with affinity to thread number $j$ will be copied to the slice number $j$ of the $i^{th}$ thread, as depicted in Figure 4.5. Therefore, in this collective, all the threads communicate with every other thread, and end up with an amount of data of the same size of the data that they had at the beginning of the operation.

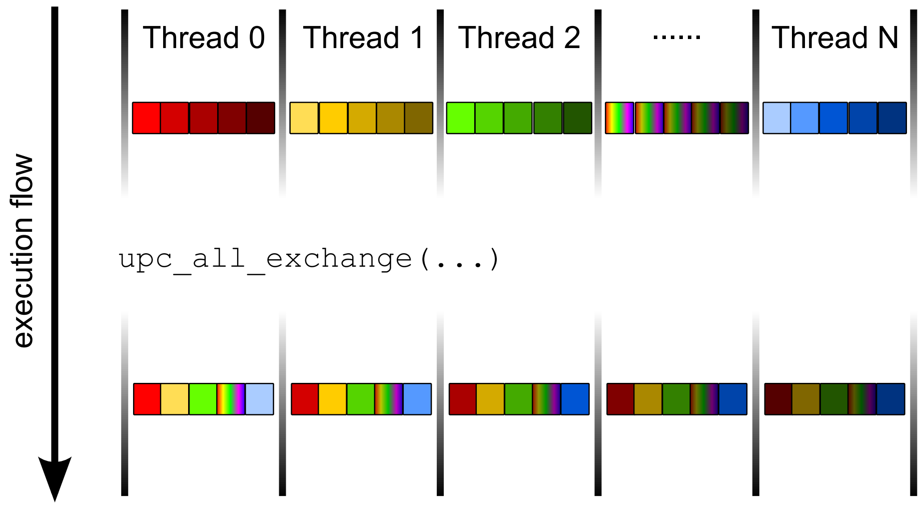


Figure 4.5: Exchange operation in UPC

The last relocalization operation is permute. Its interface is: `void upc_all_permute(shared void * restrict dst, shared const void * restrict src, shared const int * restrict perm, size_t nbytes, upc_flag_t flags);`. Permute is a very irregular collective, whose communication pattern depends enterily on the array `perm`. This way, the block of size `nbytes` with affinity to thread $i$ will be copied to the thread whose number is in $i^{th}$ position of the `perm` array. Each thread can appear only once in the `perm` array. This operation is sketched in Figure 4.6.

### 4.1.2. Computational Operations

In UPC there are two computational collective operations, reduce and prefix reduce. In reduce, the data contained in a shared array is reduced to a single

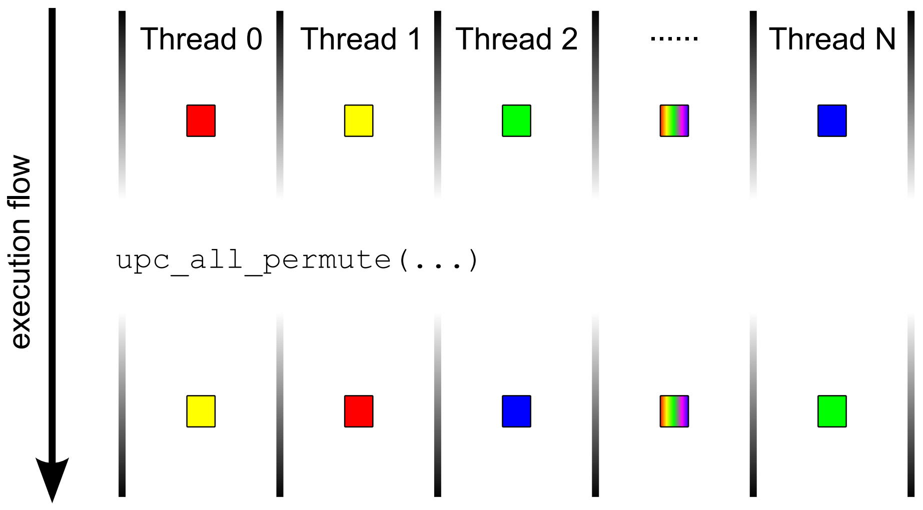


Figure 4.6: Permute operation in UPC

value. Its interface is: `void upc_all_reduce<<T>>(shared void * restrict dst, shared const void * restrict src, upc_op_t op, size_t nelems, size_t blk_size, <<TYPE>>(*func)(<<TYPE>>, <<TYPE>>), upc_flag_t flags);`, where elements between << and >>, that is <<T>> and <<TYPE>>, denote primitive numerical data types, `op` denotes an operation type and `func` is an optional user defined operation, that can be commutative or non-commutative, depending on the value of `op`. Figure 4.7 depicts an example of this operation.

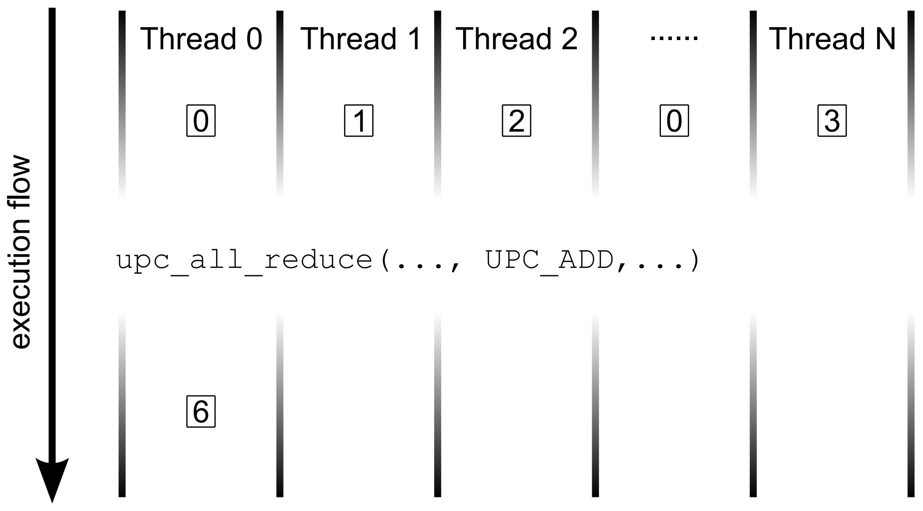


Figure 4.7: Reduce operation in UPC

Prefix reduce is an operation similar to reduce, but their differences have profound implications. Its interface is: **`void upc_all_reduce<<T>>(shared void * restrict dst, shared const void * restrict src, upc_op_t op, size_t nelems, size_t blk_size, <<TYPE>>(*func)(<<TYPE>>, <<TYPE>>), upc_flag_t flags);`**. What differentiates prefix reduce from prefix is that the result is accumulative, rather than a single value. Therefore, the destination array has the same size and distribution as the source array. For each position of the destination array, the value there contained is the result of applying a reduction to all the previous elements in the source array, as shown in Figure 4.8.

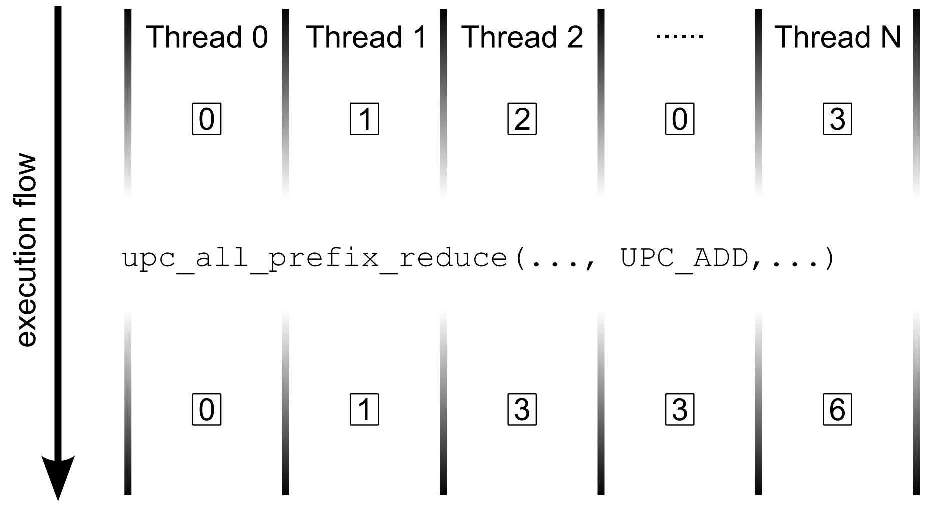


Figure 4.8: Prefix reduce operation in UPC

### 4.1.3. Data Distribution Groups

Considering data distribution, there are a clear set of collectives groups:

- One to all collectives, like broadcast and scatter.
- All to one collectives, like gather and reduce.
- All to all collectives, like gather all and exchange.
- One to one collectives, namely permute.

- Accumulative collectives, namely prefix reduce.

One to all and all to one kind of collectives are the most used and important set of collectives. These collectives can be implemented efficiently using trees for communication. The other set of collectives are less important and with more limited optimization possibilities, due to their data communication patterns. Therefore, the focus of this Thesis is largely on collectives that can be implemented with trees.

## 4.2. Design of Scalable Collective Operations for PGAS Languages

Scalability is a pervasive and complex problem in HPC. Sometimes an algorithm presents easy development and good performance but it might not scale well with hundreds or thousands of cores, hindering the use of today's HPC systems at their full power. The key issue is the use of highly scalable methods, even though they might be more complex. This kind of methods should be the preferred choice to face up large scale problems, as demonstrated in [71], whereas less complex algorithms with good performance are acceptable for small or medium scale setups. This principle is valid both for applications and collective operations libraries. The algorithms developed in this Thesis aims at scalable performance on hundreds or thousands of cores on NUMA clusters rather than providing efficiency on small/medium scale setups.

Often, high performance libraries are developed with a target architecture in mind, using specific features of that architecture. However, in the development of communication algorithms, it is highly desirable to provide portable libraries to be used by different runtimes in different systems. Therefore, one major requirement of this library was to work with as few as possible runtime specific features, and rely mostly on standardized functions of the UPC language. As a consequence, the library requires UPC standard operations and the optional libraries described in the UPC specification, present in all the current implementations. It also requires Berkeley semaphores. These semaphores allow point-to-point synchronization, a must-have for scalable non-system-wide synchronization. Even though the semaphores interface is not part of the specification, the main implementation –i.e. Berkeley UPC– supports them. Hewlett-Packard has provided semaphores for their HP UPC implementation, even though they are not part of the official distribution. Besides this, semaphores can be implemented as wrappers around some standard UPC constructs. Keeping this in mind, the conclusion is that the software stack depicted in Figure 4.9 is portable to every major UPC implementation.

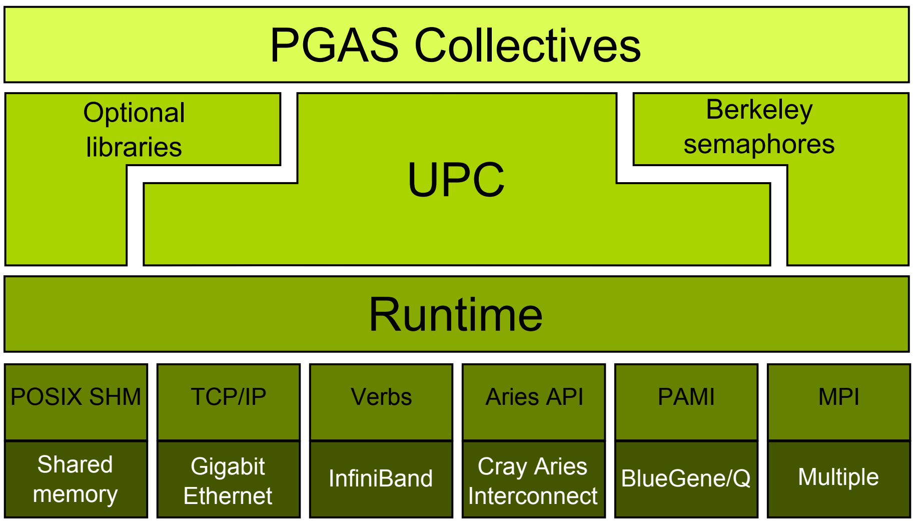


Figure 4.9: Software stack of the developed PGAS collectives

Validation of results is a key aspect of high performance and scientific computing. In order to ensure the correctness of collective operations libraries, the George Washington University has released a testing suite called GUTS (GWU Unified Testing Suite) [24]. This suite has been used during the development of the algorithms proposed in this Thesis.

### 4.2.1. Broadcast and Baseline Algorithm

Traditionally, the most efficient collective implementations use trees of processes to distribute or gather the data, although generally regardless the processes placement. Only some advanced solutions implement topology or multicore aware trees [45, 46, 50, 113]. The algorithm presented in this Thesis extends these approaches to NUMA clusters, taking into account the NUMA topology. Therefore, the trees used will be decomposed in three levels of subtrees: (1) the cluster level, (2) the node level, and (3) the NUMA region level.

Figure 4.10 illustrates the proposed structure for the algorithm using an example which consists of 8 nodes, each one of them with 24 cores distributed between 4 NUMA regions. The proposed algorithm will be able to distribute effectively and efficiently the data transfers among processes, taking advantage of the increased locality at the same time, as it minimizes the usage of the most costly links, using the fastest data channels whenever is possible, taking the most out of runtimes' shared memory optimizations. In fact, even runtimes without shared memory optimizations can get an extra benefit. For instance a UPC runtime without shared memory optimizations but support for privatizability functions [120] can map page tables from other processes into its own memory space, allowing the use of the much faster `memcpy` system call.

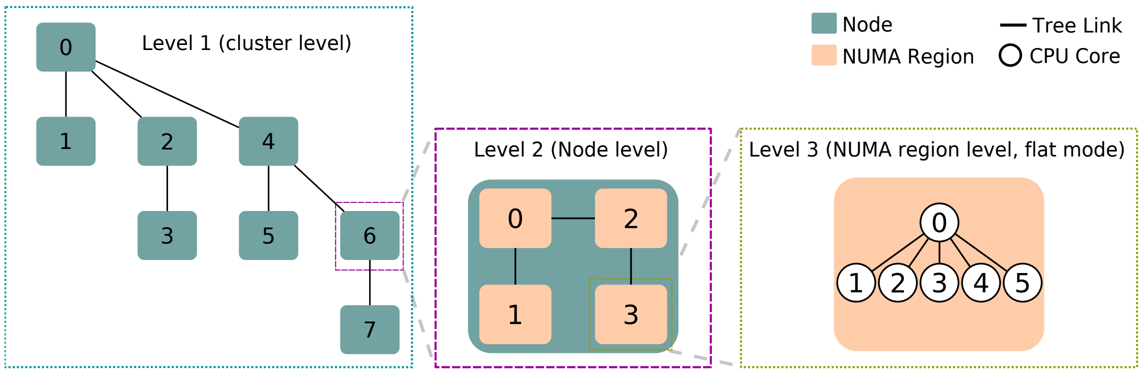


Figure 4.10: General overview of the scalable algorithm for collective operations on NUMA clusters

One of the design principles for scalability is to avoid the use of dynamic structures whose size and build time overhead increases with the number of processes. Thus, in the proposed algorithms the first call to a collective function creates a persistent and fixed (invariable) process tree structure, which can be reused in a future collective call. If the root of the collective operation and the root of the tree are not the same, then a copy of the message into the tree root is required, in top-down operations like scatter or broadcast, or the copying from the tree root to the operation root, in bottom-up operations like gather. This approach has as main benefit the reuse of the tree structures through all the run time. However, for a reduced number of processes it is still faster building a custom tree than reusing a structure, if the root of the collective operation is not the root of the precomputed tree.

In the cluster level the nodes are interconnected through a network. For each node one process is selected as a node leader, in charge of communicating its node with the other nodes. Under certain circumstances it would be desirable to have multiple node leaders (for instance, in systems with more than one network interface per node). However, most systems still have one high-speed network interface, such as the one analyzed in this Thesis, so generally the number of node leaders would be one. For efficiency and scalability purposes, a binomial tree of node leaders will be built.

The node level comprises the NUMA regions available in a node. The consideration of this level is one of the contributions of this Thesis. For each NUMA region one process is selected as NUMA region leader, and a binomial tree of NUMA region leaders will be built, with the node leader as root of its node tree. This level leader will be responsible for the communication of its children and its parent, that could be the node leader or another NUMA region leader in systems with multiple NUMA regions. The tree used for this level is also a binomial tree.

The NUMA region level connects its elements through shared memory. In this level a new tree will be built, with the NUMA region leader as root. Here processes are attached to the NUMA region through binding, thus avoiding process migration to another NUMA region. Using NUMA region binding rather core binding is advisable, as it allows the operating system to move processes within the NUMA region if necessary. However, in some architectures the cache or bus sharing can have a significant impact [55]. Therefore, in these architectures a per core binding could be a better choice. In this level two types of trees have been implemented and tested: binomial and flat trees. In the proposed algorithm there is no reason to avoid a process from being leader at several levels. In fact, it is advisable. Therefore, the node leader has been made also its NUMA region leader. Systems without NUMA capabilities are treated as having a single NUMA region. Therefore, the node level behaves like the NUMA region level, which does not exist in these cases. The Intel Xeon Phi architecture is an example of such systems.

The reason for using binomial trees instead of binary trees is their reduced number of steps needed to traverse them in setups with large number of nodes. Binomial trees will complete a 1-to-N or N-to-1 operation (broadcast, gather, scatter and reduce) in a $\lceil log_2(N)\rceil$ number of steps, for an $N$ number of nodes. That is considering that the communication starts towards the deepest branch, and that the communication is done with one connected node at a time (sibling nodes can not communicate at the same time with their parent). Binary trees on the other hand, will complete the operation in $(\lceil log_2(N)\rceil - 1) * 2 - 1$ number of steps in the best case, or $(\lceil log_2(N)\rceil - 1) * 2$ in the worst case, for $N > 2$. For 16 processes the binomial tree will finish in 4 steps, whereas the binary tree will finish in 6. For 4096 nodes the difference is 12 vs. 22.

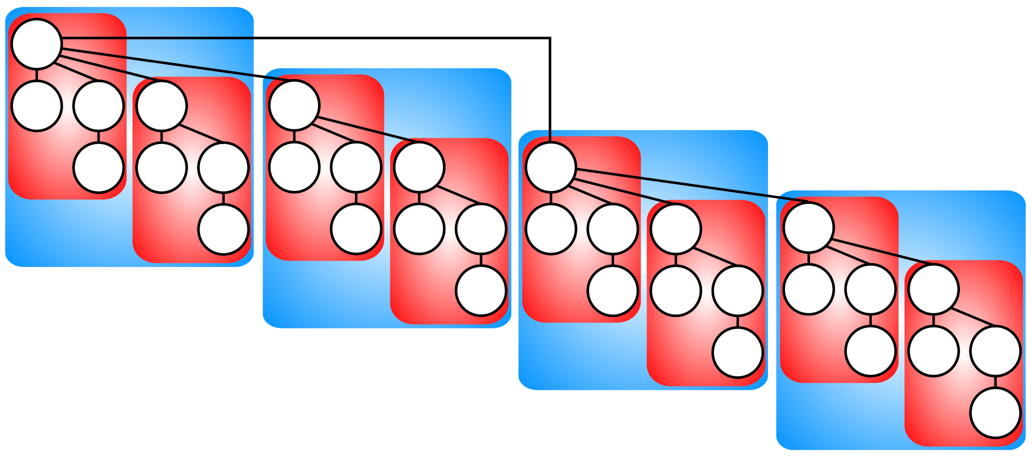
Therefore, binomial trees are a better choice for scalable communications. However, it shall be noted that binary trees can be also a valid option when communication between nodes in the tree can be non-blocking and/or one-sided. In these cases/scenarios communications can be overlapped, making the time required to communicate with all the children nodes in a lower level close to the time required to communicate with just one node. In that case the operation will be completed in a maximum of $(\lceil log_2(N)\rceil - 1)$ number of steps. This is true when both transfers can be done simultaneously without mutual interference, which is usually not possible, and is highly dependent on the bandwidth and the message rate that the network adapter can handle. Moreover, if that scenario is possible, binomial trees will also finish in a $(\lceil log_2(N)\rceil - 1)$ number of steps, which makes considering binary trees impractical in most situations.

Flat trees do not scale, as they saturate the sender or receiver (depending on the operation) easily. However, for a small number of nodes in the tree, a flat tree avoids intermediate steps, reducing the synchronization overhead. The library presented in this Thesis also evaluates the use of flat trees in the NUMA region level. Figure 4.11 illustrates the mapping of tree nodes to computing nodes and NUMA regions, using both binomial and flat trees.

In the case of computing nodes with a number of processes/threads that is power of 2, a hierarchical tree based on binomial trees will look like exactly the same as a non hierarchical tree. The mapping seen in Subfigure 4.11a would be essentially the same. However, if the number of processes/threads is not a power of 2 then the non-hierarchical tree would map differently into the hardware, having multiple connections between nodes, in a way that the use of the slowest paths is not minimized. With hierarchical trees this situation, that is becoming more common with new processors whose number of cores is not a power of 2, is avoided. Figure 4.12 illustrates this situation, using both binomial and flat trees at the NUMA region level.

Another feature present in high speed network fabrics is the presence of separate links for upload and download data. Bearing that in mind, it is possible to pipeline communications, overlapping send and receive operations to reduce latency. The proposed collective algorithms implement two fragmentation schemes for pipelining: static and dynamic. In the static mode the message is fragmented into chunks of a given size. This way, when one chunk is received, the destination process is able (if necessary) to forward that data while receiving the next chunk. This operation goes on until the complete message has been delivered. The dynamic mode is similar to the static mode, except that it splits the message in two halves, instead of $\lceil message\_size/chunk\_size \rceil$ messages. Thus the size of the chunks changes depending on the size of the message. The selected chunk size for the static mode is 32768 bytes. The dynamic mode will start fragmenting the messages when they are larger than 8192 bytes. It should be noted at this point that MVAPICH2 implements pipelining, but in a different way. In hierarchical algorithms, with differentiated steps for intra- and internode communications, MVAPICH2 can slice up the messages and perform the intra- and internode steps for every slice, rather than the whole message. However, this does not allow communication overlapping at every branch, and is a simpler and less effective mechanism.

This library also takes advantage of one-sided memory copies, implementing most functions in two approaches: *push* and *pull*. In the push approach the source process puts the data in the destination process, whereas in the pull mode it is the destination process the one which gets the data. This way it is possible to achieve a higher degree of communication overlapping, since data is streamed to/from different sources at the same time. However, it should be noted also that this library has been implemented with support for non-blocking memory transfers, also allowing a high degree of overlapping in cases where the thread that initiates the communication has to communicate with multiple threads. If the non-blocking memory transfers –"asynchronous" as denominated originally by the Berkeley UPC implementors– are not available, the blocking variant is used.


(a) Binomial trees at the NUMA region level

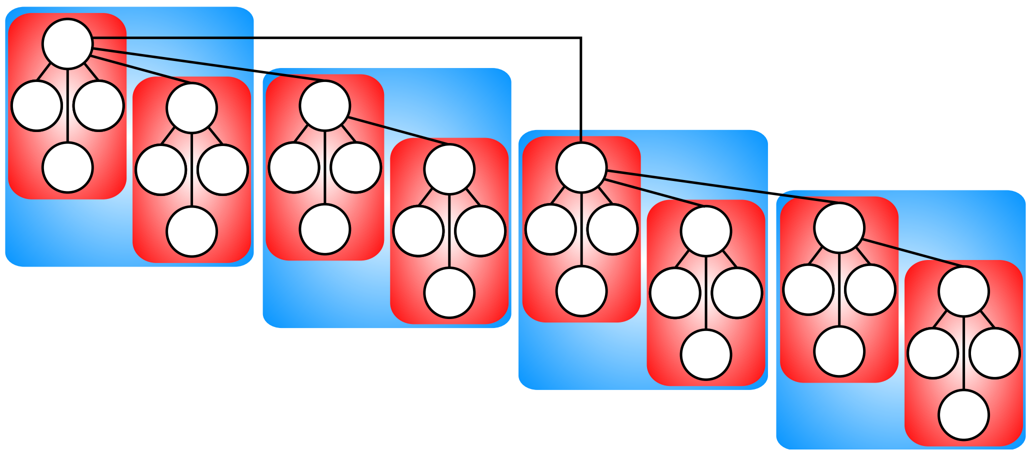
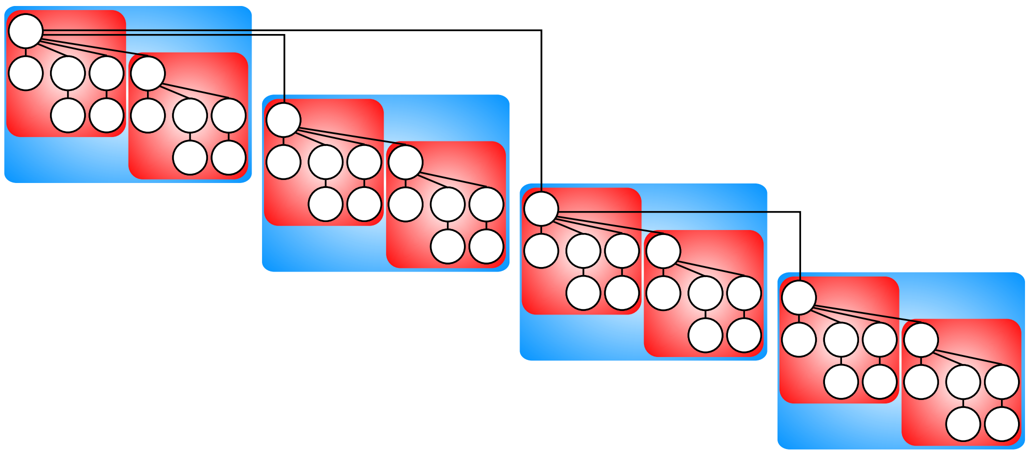
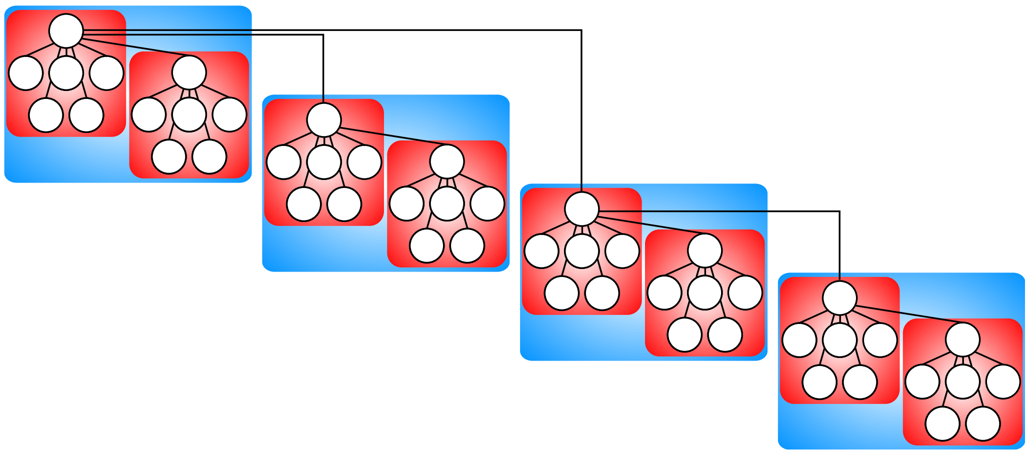

(b) Flat trees at the NUMA region level

Figure 4.11: Tree mapping with a power of 2 number of cores



(a) Binomial trees at the NUMA region level



(b) Flat trees at the NUMA region level

Figure 4.12: Tree mapping with a non power of 2 number of cores

The broadcast collective can be optimized using all the described methods. Algorithm 1 shows the pseudocode of broadcast with all the optimizations. I.e.: Using one-sided memory copies in a pull fashion, hierarchical trees and extensive message pipelining.

**Algorithm 1**: Pseudocode of broadcast algorithm with pull approach and message pipelining

```
if $\neg initialized$ then $collectives\_initialization()$
$current\_chunk \leftarrow threshold$
$number\_of\_iterations \leftarrow \lceil message\_size/threshold \rceil$
if $thread \neq 0$ then $semaphore\_wait(my\_thread, 1)$
if $thread = 0$ then
    if $upc\_castable(source)$ then $copy \leftarrow local\_get$
    else $copy \leftarrow remote\_get$
    for $i \leftarrow 0$ to $number\_of\_iterations$ do
        $copy(destination + offset, source + offset, current\_chunk)$
        for $j \leftarrow 0$ to $number\_of\_children$ do
            $semaphore\_post(tree.children[j], 1)$
        if $i + 1 = iterations - 1$ then
            $current\_chunk \leftarrow message\_size - data\_already\_sent$
else
    if $upc\_castable(destination[tree.parent])$ then $copy \leftarrow local\_get$
    else $copy \leftarrow remote\_get$ for $i \leftarrow 0$ to $number\_of\_iterations$ do
        $copy(destination + offset, destination[tree.parent] +$
        $offset, current\_chunk)$
        for $j \leftarrow 0$ to $number\_of\_children$ do
            $semaphore\_post(tree.children[j], 1)$
        if $i + 1 = iterations - 1$ then
            $current\_chunk \leftarrow message\_size - data\_already\_sent$
        if $i + 1 < iterations$ then
            $semaphore\_wait(my\_thread)$
```

### 4.2.2. Particularities of Scalable Scatter/Gather Operations

Scatter and gather operations have a particularity. In these functions the source or destination of the data, respectively, is a single process, while the other processes receive or send, respectively, their specific chunk of data. Therefore, the data movement can not be optimized in the same way as for broadcast, since each process holds a different chunk of data. For this reason, the use of trees pose more difficulties for data distribution in these functions. However, a collective using trees avoids the overhead of each process copying data separately, since less copies from/to source/destination will be done. Such approach seem interesting in scenarios where the data held by each process is not excessive. Besides this, having a scatter or gather function that uses trees has an additional benefit. Since just a few processes will communicate with the root of the operation, the memory footprint will be smaller in some systems, leading to a higher scalability. In high-speed cluster networks, such as InfiniBand, a small buffer is used for each peer connection. In jobs with thousands of processes this becomes a big problem as it has been pointed out before in several works [48, 97]. Mitigating this effect usually involves deep changes in the communication layer of the runtime or the transport layer. Shared Receive Queues (SRQ) and eXtended Reliable Connection (XRC) are recent InfiniBand features that allow to minimize the memory overhead in large setups. The UPC collectives library proposed in this Ph.D. Thesis and a runtime/driver with support for on-demand connections, where buffers are allocated as needed instead of at initialization, help to solve this problem for scatter and gather at a higher level than runtime or transport layer modifications.

The aforementioned trees take advantage of the underlying hardware and memory hierarchy. In scatter and gather operations in order to move data efficiently downward or upward the tree, the processes have to be contiguous within a given branch. This cannot be guaranteed, as the user can choose a cyclic process distribution among nodes. One possible workaround is having each level root being aware of all the processes (and their order) hanging in all their branches. However, this workaround has two major issues. The first issue is that the tree root would need to store too much information about the tree, increasing with the number of processes and thus preventing scaling due to their increasing memory footprint. The second and more important issue is that root processes would have to perform multiple out-of-order memory copies, instead of a single big memory copy. The overheads of any of these two issues make this workaround impractical.

A trade-off solution is building a binomial tree with all the processes, ignoring their distribution among nodes and NUMA regions. In an ideal case, with a block distribution and a power of 2 number of processes in the nodes and in the NUMA regions, the tree built this way would map perfectly into the hardware, as in Subfigure 4.11a, minimizing the use of the links with more latency.

Tree-based scatter and gather functions have also other particularities. They use intermediate buffers to copy data. The buffer management code is performed before the initial barriers (if `UPC_IN_ALLSYNC` or `UPC_IN_MYSYNC` are set) and the ending barriers (if `UPC_OUT_ALLSYNC` or `UPC_OUT_MYSYNC` are set). The buffers are not reallocated if the previously allocated buffers are big enough. However, if a certain call needs a buffer size of more than a certain threshold (currently $16MB$), the buffer will be freed at the end of the function to avoid excessive memory usage.

Another particularity is that process 0's buffers will be the source or the destination (in gather or scatter, respectively), if the process 0 is the root of the operation.

The last particularity is that, even though these functions do not take advantage of the processes' distribution and trees do not map onto the hardware, they are bound to the corresponding NUMA region, since this step is performed when the library is initialized, at the beginning of each runtime execution.

Algorithm 2 describes how the scatter collective has been implemented using binomial trees, one-sided memory copies with a pull approach and message pipelining.

So far the described particularities are for both scatter and gather. Gather has an extra particularity. It does not have a dynamic fragmentation version. The reason for this is that, since the data flows upwards, copying the first half do not make sense in most situations. The parent process could not take advantage of it, since its own first half will be larger than any of its children's first half. Therefore, data can not flow in halves because parent processes would have to wait for the second half anyway before sending their first half. Algorithms 3 and 4 describe how the gather collective has been implemented using binomial trees, one-sided memory copies with a push approach and static message pipelining.

**Algorithm 2**: Pseudocode of scatter algorithm with pull approach and message pipelining

```
1  if $\neg initialized$ then $collectives\_initialization()$
2  $nchunks \leftarrow calculate\_number\_of\_threads\_down\_the\_tree(...)$
3  $my\_size \leftarrow nchunks * nbytes$
4  $current\_chunk \leftarrow calculate\_chunk\_size(...)$
5  $number\_of\_iterations \leftarrow \lceil my\_size/threshold \rceil$
6  $allocate\_buffer\_if\_needed(...)$
7  $allocate\_and\_initialize\_extra\_data\_structures(...)$
8  $sent\_data \leftarrow 0$
9  for $i \leftarrow 0$ to $number\_of\_iterations$ do
10     if $my\_thread \neq 0$ then $semaphore\_wait(my\_thread)$
11     if $upc\_threadof(source) \neq my\_thread$ then
12         if $upc\_castable(source)$ then $copy \leftarrow local\_get$
13         else $copy \leftarrow remote\_get$
14         $copy(destination + offset, source + offset, current\_chunk)$
15     $sent\_data$ += $current\_chunk$
16     if $nchunks > 1$ then
17         $copied\_chunks \leftarrow calculate\_copied\_chunks(...)$
18         $child\_id \leftarrow 0$
19         while $copied\_chunks > 0$ do
20             if $remaining\_chunks[child\_id] > 0$ then
21                 $semaphore\_post(tree.children[child\_id], 1)$
22                 $remaining\_chunks[child\_id] - -$
23                 $copied\_chunks - -$
24             else
25                 $child\_id + +$
26                 if $child\_id \geq tree.num\_children$ then $break$
27     $compute\_next\_chunk\_and\_offsets(...)$
28 $semaphore\_post(tree.parent, 1)$
29 $free\_extra\_data\_structures(...)$
30 if $nchunks > 1$ then
31     $copy\_data\_to\_own\_destination(...)$
32     if $buffer\_size > max\_cacheable\_buffer$ then
33         $semaphore\_wait(my\_thread, tree.num\_children)$
34         $free\_buffer(...)$
```

**Algorithm 3**: Pseudocode of gather algorithm with push approach and message pipelining (1 of 2)

```
1  if $\neg initialized$ then $collectives\_initialization()$
2  $allocate\_buffer\_if\_needed(...)$
3  for $j \leftarrow 0$ to $number\_of\_children$ do
4      $semaphore\_post(tree.children[j], 1)$
5  $nchunks \leftarrow calculate\_number\_of\_threads\_down\_the\_tree(...)$
6  $my\_size \leftarrow nchunks * nbytes$
7  $current\_chunk \leftarrow calculate\_chunk\_size(...)$
8  $number\_of\_iterations \leftarrow \lceil my\_size/threshold \rceil$
9  $allocate\_and\_initialize\_extra\_data\_structures(...)$
10 if $my\_thread \neq 0$ then $semaphore\_wait(my\_thread)$
11 if $leaf\_thread(my\_thread) = true$ then
12     for $i \leftarrow 0$ to $number\_of\_iterations$ do
13         if $upc\_castable(destination)$ then $copy \leftarrow local\_get$
14         else $copy \leftarrow remote\_get$
15         $copy(destination + offset, source + offset, current\_chunk)$
16         $semaphore\_post(tree.parent[my\_child\_id], 1)$
17         $compute\_next\_chunk\_and\_offsets(...)$
18
```

**Algorithm 4**: Pseudocode of gather algorithm with push approach and message pipelining (2 of 2)

```
19 else
20     $my\_iters \leftarrow nbytes/threshold$
21     for $i \leftarrow 0$ to $my\_iters$ do
22         if $upc\_castable(destination)$ then $copy \leftarrow local\_get$
23         else $copy \leftarrow remote\_get$
24         $copy(destination + offset, source + offset, threshold)$
25         $semaphore\_post(tree.parent[my\_child\_id], 1)$
26         $compute\_offsets(...)$
27     if $nbytes\%threshold \neq 0$ then
28         if $upc\_castable(destination)$ then $copy \leftarrow local\_get$
29         else $copy \leftarrow remote\_get$
30         $copy(destination + offset, source + offset, nbytes\%threshold)$
31         $compute\_offsets(...)$
32     $current\_chunk \leftarrow calculate\_chunk\_size(...)$
33     $current\_child \leftarrow 0$
34     if $\neg(my\_thread = 0 \&\& upc\_threadof(dst) = 0)$ then
35         for $i \leftarrow my\_iters$ to $number\_of\_iterations$ do
36             if $current\_child < number\_of\_children$ then
37                 $calculate\_current\_child\_chunks(...)$
38                 while $acum < threshold$ do
39                     $semaphore\_wait(my\_thread[current\_child])$
40                     $calculate\_acummulated\_and\_current\_child(...)$
41                     if $last\_chunk$ then break
42             if $upc\_castable(destination)$ then $copy \leftarrow local\_get$
43             else $copy \leftarrow remote\_get$
44             $copy(destination + offset, source + offset, current\_chunk)$
45             $semaphore\_post(tree.parent[my\_child\_id], 1)$
46             $compute\_next\_chunk\_and\_offsets(...)$
47 if $nchunks > 1$ then
48     if $buffer\_size > max\_cacheable\_buffer$ then
49         $free\_buffer(...)$
50 $free\_extra\_data\_structures(...)$
```

Lastly, an additional algorithm has been implemented for scatter and gather. The tree structure is not appropriate when the chunks of data are too big. A token-passing algorithm has been developed for those cases. The token is passed to the next process, in a ring fashion. The process with the token starts copying data. This way the algorithm prevents that all process access at the same time, saturating the network. The token is passed to the next process when one of the following conditions are met: (1) the current process is in the same node as the source/destination process, before start copying, to allow overlapping using the fast memory subsystem; (2) the data to be copied is smaller than a given threshold, to avoid the following processes wait unnecessarily; (3) when the remaining data to be transferred is smaller than the previous threshold. When the data is bigger than this threshold the copy is performed in two phases, the first one with a size $N - Threshold$ and the second one with a size $Threshold$. Since the bottleneck of scatter and gather operations is the outbound link from the source thread, the presented implementation operates with a single token, assuming that a single thread can saturate the network or, when this condition is not met, passing the token before initiating the copy, to allow overlapping of copies to/from different threads. Operations with very small messages and a large number of threads can benefit from the tree algorithm or use a ring algorithm with multiple tokens. Nevertheless, it should be noted that even though the ring algorithms fit the semantics of the scatter and gather operations, the fact that all the processes/threads have to communicate with a single root can impose some scalability issues, in particular regarding memory footprint of the communication buffers. Algorithms 5 and 6 describe the ring algorithms for scatter and gather respectively.

### 4.2.3. Particularities of Scalable Reduce Operations

The semantics of the Reduce operation in UPC is different from the behavior of the Reduce operation in MPI. In UPC, all the values of a shared array are reduced to a single element, as opposed to MPI, where the result is an array, with reduced values for every array position. Therefore, the developed algorithms do not conform the definition of the reduce operation in MPI, and no comparison between the two will be made.

**Algorithm 5**: Pseudocode of scatter algorithm with pull approach on a ring

```
if ¬$initialized$ then $collectives\_initialization()$
if $my\_thread = upc\_threadof(src)$ then
    if $my\_thread \neq 0$ then $semaphore\_post(0, 1)$
    else $semaphore\_post(next\_thread, 1)$
    $memcpy(my\_dst, my\_src, nbytes)$
else
    $semaphore\_wait(my\_thread, 1)$
    if $upc\_castable(src)$ then
        $semaphore\_post(next\_thread, 1)$
        $memcpy(my\_dst, my\_src, nbytes)$
    else
        $first\_chunk \leftarrow calculate\_first\_chunk()$
        $second\_chunk \leftarrow calculate\_second\_chunk()$
        if $first\_chunk > 0$ then $upc\_memget(my\_dst, my\_src, first\_chunk)$
        if $next\_thread \neq upc\_threadof(src)$ then
            $semaphore\_post(next\_thread, 1)$
        else if $next\_thread + 1 < THREADS$ then
            $semaphore\_post(next\_thread + 1, 1)$
        $upc\_memget(my\_dst + first\_chunk, my\_src + first\_chunk, second\_chunk)$
```

**Algorithm 6**: Pseudocode of gather algorithm with push approach on a ring

```
1  if $\neg initialized$ then $collectives\_initialization()$
2  if $my\_thread = upc\_threadof(dst)$ then
3      if $my\_thread \neq 0$ then $semaphore\_post(0, 1)$
4      else $semaphore\_post(next\_thread, 1)$
5      $memcpy(my\_dst, my\_src, nbytes)$
6  else
7      $semaphore\_wait(my\_thread, 1)$
8      if $upc\_castable(src)$ then
9          $semaphore\_post(next\_thread, 1)$
10         $memcpy(my\_dst, my\_src, nbytes)$
11     else
12         $first\_chunk \leftarrow calculate\_first\_chunk()$
13         $second\_chunk \leftarrow calculate\_second\_chunk()$
14         if $first\_chunk > 0$ then $upc\_memput(my\_dst, my\_src, first\_chunk)$
15         if $next\_thread \neq upc\_threadof(src)$ then
16             $semaphore\_post(next\_thread, 1)$
17         else if $next\_thread + 1 < THREADS$ then
18             $semaphore\_post(next\_thread + 1, 1)$
19         $upc\_memput(my\_dst + first\_chunk, my\_src +$
           $first\_chunk, second\_chunk)$
```

The developed reduce function is based on several design principles looking for scalability. The first one is that each process performs the reduction of its own data. Therefore, the data communication is restricted to communicating a single element of a primitive data type, that is to say, from a minimum size of `char` and a maximum size of `long double`. Due to this, no fragmentation occurs.

**Algorithm 7**: Pseudocode of reduce algorithm with push approach

```
1  if $\neg initialized$ then $collectives\_initialization()$
2  $calculate\_alignment\_and\_offsets(...)$
3  $reduce(own\_data)$
4  $semaphore\_wait(my\_thread, tree.number\_of\_children)$
5  $reduce(own\_data\_and\_children\_data)$
6  if $my\_thread \neq tree\_root$ then
7      my_destination = reduce_buffer[tree.parent]
8  else
9      $my\_destination \leftarrow destination$
10 if $upc\_castable(my\_destination)$ then
11     $local\_put(my\_local\_destination + offset, local\_reduce\_buffer[my\_thread], size\_of(reduction\_type))$
12 else
13     $remote\_put(my\_destination + offset, local\_reduce\_buffer[my\_thread], size\_of(reduction\_type))$
14 if $my\_thread \neq tree\_root$ then
15     $semaphore\_post(tree.parent, 1)$
```

The second consideration is motivated by the fact that a process might not know if its children participate in the reduction. To solve this issue, each process with a passive participation will contribute to the operation with a neutral operand value (e.g., 0 for add operations and 1 for multiplications). In case the user defines its own operation then this value must be adapted.

The third reduce design principle is a consequence of the tree used. Thus, for noncommutative operations (such as noncommutative operations defined by the user, `UPC_NONCOMM_FUNC`), the operations must take the order into account, otherwise they will provide an erroneous result.

The use of non-topology-aware binomial trees supports the two first design principles. However, this would neglect the benefits of hierarchical trees. Algorithm 7 illustrates the pseudocode of the reduce algorithm with one-sided communication with push approach.

### 4.2.4. Summary of the Implemented Algorithms

The number of variations of the developed algorithms is a result of combining different orthogonal optimizations suited for the operations. Table 4.1 presents an overview of the developed algorithms. It should be noted that, even though these algorithms have been implemented in UPC, the underlying principle and optimizations are also valid for MPI. However, UPC, and more generally any PGAS approach, allows to implement them in a more natural way, since one-sided communication is a key feature of the model. In MPI, put and get operations, and their non-blocking counterparts, require explicit memory and window management, as opposed to UPC.

Table 4.1: Summary of the Optimized PGAS Collective Algorithms Implemented. Scatter and gather collectives with tree-based algorithms use normal binomial trees with binding.

| | | | Operations | | | |
|---|---|---|---|---|---|---|
| | | | Broadcast | Reduce | Scatter | Gather |
| Push | Ring | | | | ✓ | ✓ |
| | Hierarchical binomial | Standard | ✓ | ✓ | ✓ | ✓ |
| | | Dynamic fragmentation | ✓ | | | |
| | | Static fragmentation | ✓ | | | ✓ |
| | Hierarchical binomial+flat | Standard | ✓ | ✓ | | |
| | | Dynamic fragmentation | ✓ | | | |
| | | Static fragmentation | ✓ | | | |
| Pull | Ring | | | | ✓ | ✓ |
| | Hierarchical binomial | Standard | ✓ | ✓ | ✓ | ✓ |
| | | Dynamic fragmentation | ✓ | | ✓ | |
| | | Static fragmentation | ✓ | | ✓ | |
| | Hierarchical binomial+flat | Standard | ✓ | ✓ | | |
| | | Dynamic fragmentation | ✓ | | | |
| | | Static fragmentation | ✓ | | | |

### 4.2.5. Gather all, Exchange, Permute and Prefix Reduce Optimization

Gather all, exchange, permute and prefix reduce collectives do not naturally fit in a tree structure. Gather all and exchange imply all-to-all communications. Permute is a multiple point-to-point communication. Prefix reduce progressively accumulates data in a sequential way that do not fit in a tree. Nevertheless, from a NUMA and multi-core standpoint some optimizations could be accomplished.

#### Gather all

The gather all collective is basically a gather followed by a broadcast. However, it has been implemented in such a way that no synchronization occurs between all the threads. When a leaf thread finishes its gather phase it waits for its NUMA node root to post its primary semaphore to pull the broadcast data, avoiding this way barriers or other wide-scale synchronization mechanisms.

#### Exchange

For the exchange collective, with small data transfers, a simple algorithm was implemented. Each thread operates independently. If the destination thread for a given transfer is in the same node and the destination address is "castable" the `memcpy` function is used to perform the transfer. Otherwise, an asynchronous transfer is performed, using `upc_memput_async` (the precursor to the non-blocking variant of `upc_memput` present in the UPC specification 1.3, and proposed by Berkeley), to allow overlapping several communications. This algorithm is the more natural way of implement this operation.

However, in order to exchange larger messages it is desirable to pack all the outgoing communications to a given thread in just one transfer. This is possible in the exchange operation because all the outgoing communications to a specific thread will write data in a contiguous memory region. With this approach a large message protocol was implemented. The threads that are leaders at the node level try to allocate enough memory for the whole run-time of the collective, and free it when they finish iterating over all the threads. If this is not possible, because of the high memory requirements, they will allocate memory for one iteration and recycle it for the following iterations. Both approaches imply synchronization between local threads. Once the buffer is ready to receive data, the node root thread posts its local threads semaphores, so they can begin transferring their data to the intermediate buffer. After the copy, each non-node-root thread posts its node root thread secondary semaphore, and wait until it post its primary semaphore to signal the beginning of the next iteration. With the buffer filled with the current iteration data, the node root thread performs an asynchronous transfer to overlap communications. If the buffer has to be recycled the node root thread will wait for the handler to complete before signaling the local threads the beginning of the next iteration.

### Permute

Permute is an operation where one thread transfers data to a single different thread in an exclusive manner. I.e., no other thread can transfer data to that thread. Therefore this collective does not fit in a tree structure and the only available optimization, besides using `memcpy` for "castable" addresses, is to pack communications. But the destination thread is application-dependent, so there is no-pattern and packaging will introduce an important synchronization and buffering overhead. For that reason packaging would be desirable in very specific cases, being detrimental in most of the other cases. Therefore packaging was not included in Permute.

### Prefix reduce

Although the prefix reduce function might look like a variation of the reduce function, from the parallelization standpoint it is not. Each step of the reduction has to be stored. That prevents the use of a tree based structure. The developed algorithm is divided in three steps. First, a node local reduction is performed. Then a reduction including data from other nodes takes place. Finally, the last step propagates the node-local result to all the threads in the node. The second step is based on the dissemination barrier algorithm proposed by Mellor-Crummey and Scott [68]. Berkeley UPC's solution is also based on this algorithm, but without dividing the operation in different steps.

The first step consists in copying the source array to the destination array. This is a local step where no communication is involved, and is therefore performed by each thread with the `memcpy` function. After that, all threads will iterate over their blocks.

The next step is the node local reduction. At this point two approaches have been implemented. One for the reductions with small blocks and the other one for reductions with large blocks.

In the first algorithm the NUMA leader threads will compute the local reductions of all their children. This implies a synchronization at the beginning of the process, to ensure that all threads have finished their copy operation. This synchronization is achieved with the use of semaphores. After this initial synchronization stage the NUMA leader threads reduce in order their data and their children data, posting their parent semaphore afterwards. When all the NUMA leader threads finish their reduction, the node leader threads post the semaphore of the first NUMA leader in the node, so it can reduce the last element –and not the others, in order to propagate locally the results as fast as possible–. After the propagation of the result takes place, the NUMA leaders calculate their remaining elements. After that, each NUMA leader thread posts the node leader semaphores.

The second algorithm is more simple. Each thread in a given node computes its local reduction. Then, all the threads in that node, except the node root thread, which is the thread with the lower id, wait until the previous thread posts its semaphore. When the threads leave the wait status they compute the reduction of their last element, post the next thread semaphore and perform the reduction of their other elements.

At this point the nodes have their local reduction completed. Then an algorithm similar to dissemination barrier is performed between the nodes. For N nodes $\lceil log_2(N) \rceil$ iterations take place. In every iteration each node root thread takes the locally reduced data from just one node. The node root thread of node $i$ has target node $i - 2^j$, being $j$ the current iteration (possible values for the current iteration range from 0 to $\lceil log_2(N) \rceil - 1$). In each iteration each node root thread reduces its last element with the remote thread data, so in the next iteration it can provide the other node the globally reduced data up to that moment. Synchronization is performed using semaphores. This algorithm is illustrated in Figure 4.13.

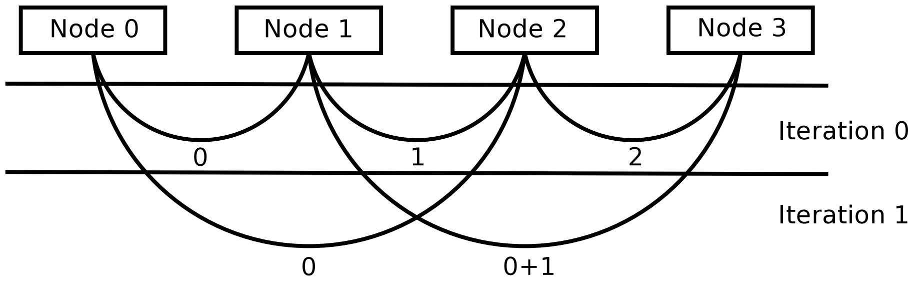


Figure 4.13: Dissemination barrier algorithm for prefix reduce operation

In the last step the remote reduced data is propagated to all the elements in the node, and is performed by NUMA node root threads for small operations, or by each thread for larger operations, in a similar way than the local reduce computation.

## 4.3. Conclusions of Chapter 4

This Chapter has presented the design of a PGAS collective library. The library has been developed from scratch, with portability, scalability and modern NUMA architectures in mind. The portability of the library has been ensured by relying in high level functions provided by the UPC specification, and semaphores outside of the specification but readily available in the main UPC implementations. The proposed design addresses scalability, by taking advantage of one-sided communication, fixed and hierarchical NUMA-aware trees, process/thread binding and a strong focus on message pipelining. Up to know no library has implemented collectives operations using all these optimization techniques. The algorithms designed aims for good scalability for current large scale systems, sacrificing performance in small setups. Moreover, even though NUMA features are a central point in the design, the algorithms can easily adapt to non-NUMA architectures, with the trees being built with a single NUMA region per node.

Some operations cannot be implemented efficiently using trees. Optimization of these functions is possible. However, the possible optimizations are not as sophisticated as the ones used for tree-based collectives. Besides this, the tree-based collectives –broadcast, scatter, gather and reduce– are by far the most used collectives in HPC. Thus, the next two chapters present a performance evaluation of the optimized algorithms, focused on the tree-based collectives. Chapter 5 analyzes the performance of the developed algorithms in 5 different NUMA systems. Later on, Chapter 6 explores the performance of these algorithms in a manycore environment.

# Chapter 5

# Performance Evaluation of PGAS Collectives on NUMA Systems

The most significant collectives have been evaluated using UOMS and its performance and their subsequent analysis is presented in this Chapter. The performance of the proposed algorithms has been assessed in a wide variety of high performance architectures, using five different NUMA systems, with various architectures, ranging from departmental clusters to large scale supercomputers. The influence of different optimizations has been also assessed, and the results have been compared to MPI on the largest scale system. Section 5.1 explains the setup and architecture of the different systems. Sections 5.2, 5.3, 5.4 and 5.5 analyze the results of the broadcast, reduce, scatter and gather collectives, respectively. Section 5.6 compares the results with those of a leading MPI implementation (ParaStationMPI, based on MPICH2), using a large cluster. Section 5.7 analyzes the impact of the different optimizations implemented in the algorithms. Section 5.8 summarizes the analysis of the results.

## 5.1. Experimental Configuration

This performance evaluation has been carried out on five representative systems. The first one is the Finis Terrae supercomputer [107], composed of 142 HP Integrity rx7640 nodes, each one with 8 Montvale Itanium 2 dual-core processors (16 cores per node) at 1.6 GHz and 128 GB of memory. The processors are distributed between 2 cells, each one with 4 processors (8 cores) and its own I/O subsystem. Each cell is an independent NUMA region. The interconnection network is InfiniBand 4X DDR (16 Gbps of theoretical effective bandwidth), with Mellanox InfiniHost III Ex HCAs and a Voltaire Grid Director ISR 2012 switch. The HCAs are plugged in the cell 0. The node architecture is depicted in Figure 5.1. The node root thread is bound to the cell 0. The library has been tested with up to 1024 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/16 \rceil$, being $n$ the total number of cores used. The UPC compiler and runtime is Berkeley UPC 2.12.1, relaying on the effective the InfiniBand Verbs library for distributed memory communication. The GASNet PSHM (GASNet inter-Process SHared Memory) optimization has been enabled. The backend C compiler available in the system is the Intel 11.1.

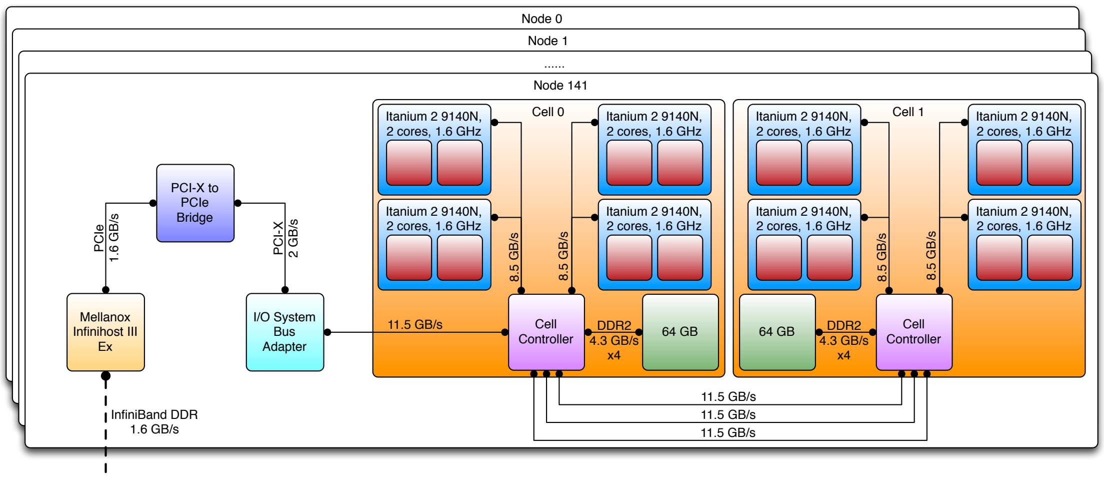


Figure 5.1: Finis Terrae node architecture

The second system is an HP Integrity Superdome at CESGA, with 64 Montvale Itanium 2 dual-core processors (128 cores total) at 1.6 GHz and 1 TB of memory. The processors are distributed between 16 NUMA regions, each one with 4 processors (8 cores). Figure 5.2 shows its architecture. The library has been tested with up to 128 cores in this system. The UPC compiler and runtime is Berkeley UPC 2.12.1, with the SMP conduit, which uses shared memory constructs for communications. The backend C compiler is the Intel 11.1.

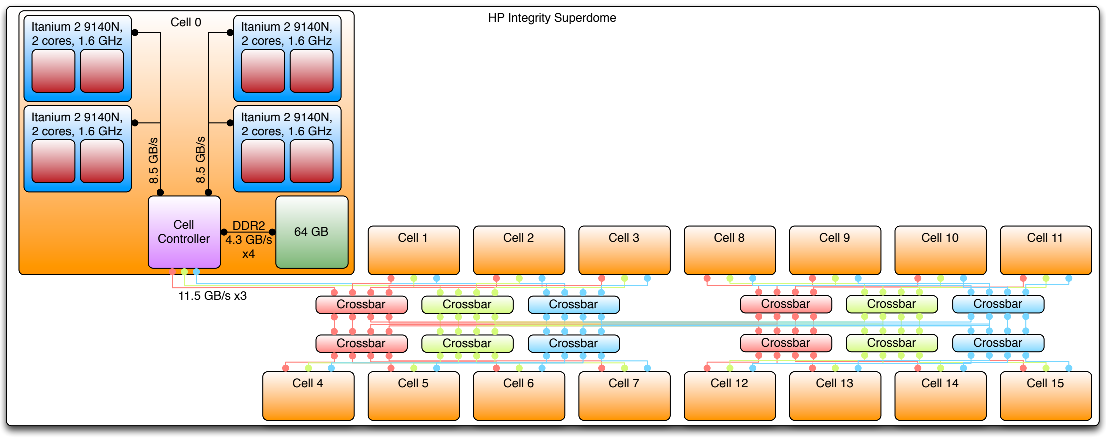


Figure 5.2: Superdome node architecture

The third system is the SVG 2011 (Galician Virtual Supercomputer) at CESGA, composed of 46 HP ProLiant SL 165z G7 nodes, each one with two 12-core AMD Opteron 6174 Magny-Cours processors (hence 24 cores per node) at 2.2 GHz, and 32 or 64 GB of memory. Each processor has 2 memory controllers. Therefore each node has 4 NUMA regions, connected through high-speed HyperTransport links. The interconnection network is Gigabit Ethernet, with HP NC362i cards. There are two interfaces bound together. A block diagram of the node architecture is presented in Figure 5.3. The bonding mode is 0 (round-robin balancing). The Ethernet switches are HP ProCurve 2910al. The library has been tested with up to 192 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/24 \rceil$, being $n$ the total number of cores used. The UPC compiler and runtime is Berkeley UPC 2.12.1, relying on the MPI conduit for distributed memory communication. Therefore the remote memory operations are built on top of MPI. In this particular testbed the implementation used is MPICH 1.3.2. The GASNet PSHM optimization has been also enabled in this system. The backend C compiler available in the system is the Open64 4.2.4.

The fourth system used in the evaluation is the JUDGE (Jülich Dedicated GPU Environment) supercomputer [108], comprised of 206 IBM System x iDataPlex dx360 M3 nodes, each one with two 6-core Intel Xeon X5650 Westmere processors (hence 12 cores per node) at 2.66 GHz, and 96 GB of memory. Each processor has its own memory controller. Therefore each node has 2 NUMA regions, connected

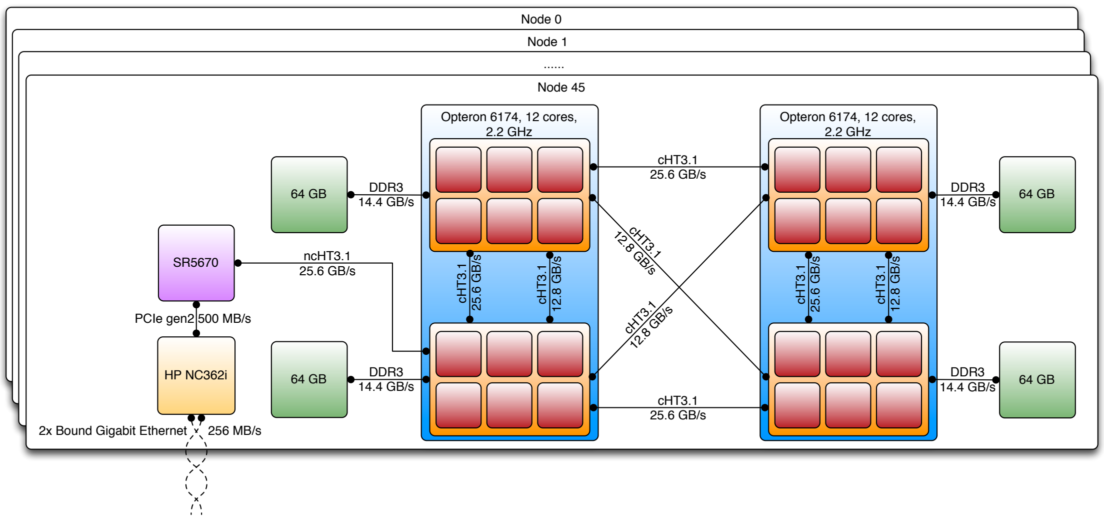


Figure 5.3: SVG 2011 node architecture

through a high-speed Intel QPI (Quick Path Interconnect) links. The interconnection network is InfiniBand 4X QDR (32 Gbps of theoretical effective bandwidth), with Mellanox ConnectX HCAs. The node architecture is presented in Figure 5.4. The InfiniBand switches are Voltaire Grid Director 4036. The library has been tested with up to 648 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/12 \rceil$, being $n$ the total number of cores used. The UPC compiler and runtime is Berkeley UPC 2.12.2, a minor release fixing some bugs on 2.12.1, and the communication layer uses the InfiniBand Verbs library for distributed memory communication. The GASNet PSHM optimization has been also enabled in this system. The backend C compiler available in the system is the Intel 11.1.

The fifth and last system is the JuRoPA (Jülich Research on Petaflop Architectures) supercomputer [109], comprised of 2208 Sun Blade 6048 nodes, each one with 2 quad-core Intel Xeon X5570 Nehalem-EP processors (hence 8 cores per node) at 2.93 GHz, and 24 GB of memory. Each processor has its own memory controller. Therefore each node has 2 NUMA regions, connected through a high-speed Intel QPI links. The interconnection network is InfiniBand 4X QDR (32 Gbps of theoretical effective bandwidth), with Mellanox ConnectX HCAs and a Sun Data Center Switch 648. The node architecture is very similar to the node architecture of JUDGE, and is depicted in Figure 5.5. The library has been tested with up to 4096 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/8 \rceil$, being $n$

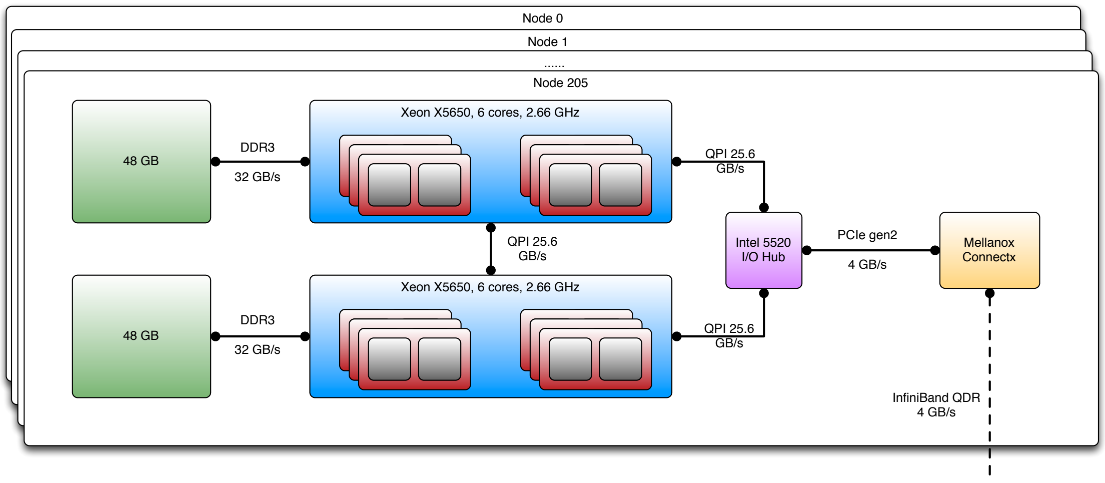


Figure 5.4: JUDGE node architecture

the total number of cores used. The UPC compiler is Berkeley UPC 2.12.2, relying on the InfiniBand Verbs library for distributed memory communication. The GASNet PSHM optimization has been also enabled. The backend C compiler available in the system is the Intel 11.1.

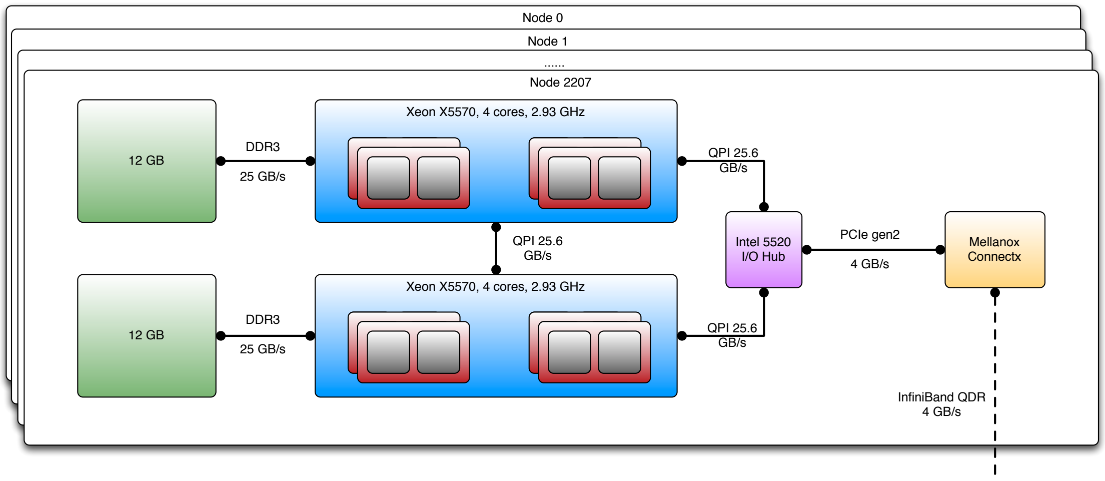


Figure 5.5: JuRoPA node architecture

Berkeley UPC offers the possibility of implementing UPC threads as POSIX threads (pthreads) or as processes. The experience gathered in Chapter 2 have shown that in order to obtain the best network access performance, pthreads should be avoided, as multiple threads accessing the UPC runtime cause contention and decrease the performance. Therefore, this evaluation has relied on UPC threads implemented as processes, with PSHM optimization to speedup the communication performance between processes in the same node.

The implementation of the proposed algorithms (from now on PGASCol) has been tested against the Berkeley UPC collectives (from now on BerkeleyCol), based on the high performance layer GASNet which implements an optimized binomial tree scheme; and also against the Michigan Technological University (MTU) reference implementation [70] of the collective operations (from now on MTUCol), based on flat trees on top of `upc_memcpy` operations. The reference implementation has, like PGASCol, two approaches: pull and push, where data is either pulled from the destination thread or pushed from the source thread.

The software used for the performance evaluation is the UPC Operations Microbenchmarking Suite (UOMS) [60], version 1.1. For each particular test, given a number of cores and message block sizes, it performs several iterations, from 1000 iterations to 20 iterations depending on the number of cores being used and the message size, to ensure representativeness and significance of the measures. All the tests have been performed in the same batch job, one after the other, to try to guarantee fairness in the comparison. Finally, the metric shown is the best result for each setup. By showing the minimum runtime the performance of the algorithms is presented without the influence of external factors (e.g., network contention/congestion), allowing to focus on the scalability of the operations that implement the proposed algorithm.

The UPC threads distribution has been performed in a block fashion. That is, consecutive thread ranks will be in the same node until the node is fully populated. This benefits algorithms that use trees but are not topology aware.

The following sections presents the performance results of four representative collectives, broadcast, reduce, scatter and gather through Figures 5.6 to 5.25. Their results have been also analyzed comparatively against MPI collectives. All figures present the performance of a representative medium-size message (16KB) on the top and the performance of a representative large-size message on the bottom. The size of the large message is 1MB for broadcast and reduce, and 64KB for scatter and gather, due to their higher memory requirements in the root process, which is the result of multiplying the message size by the number of processes. The $y$ axis represents latency in microseconds in the graphs on the top (medium-size message case), whereas the $y$ axis represents bandwidth in GB/s in the graphs on the bottom (large-size message case), except for broadcast on the SVG, which shows MB/s, and the reduce operation, which always shows latencies.

Variations in the same basic algorithm can lead to some dramatic performance differences, as shown in Table 5.1. Therefore, the graphs display only the most relevant algorithms for each combination of system, function and message size.

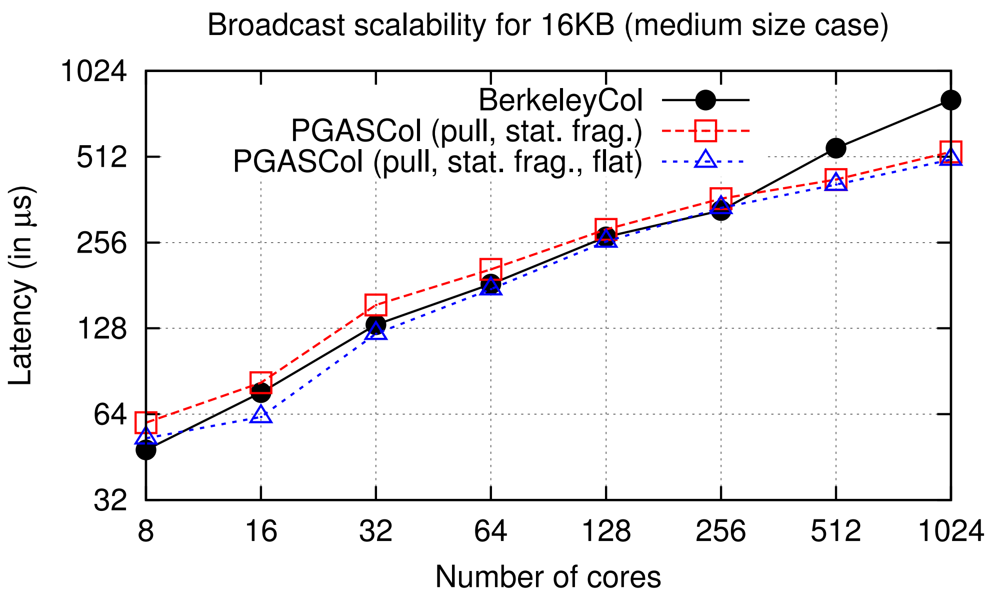
Table 5.1: Bandwidth (in MB/s) obtained by the basic pull approach algorithm (labelled Pull) and the pull algorithm with static fragmentation and flat trees at the intra NUMA level of this library (labelled Pull-s-f). The data displayed has been obtained with the maximum number of processes tested in each system. That is: 1024 cores in Finis Terrae, 128 cores in Superdome, 192 cores in SVG, 648 cores in JUDGE and 4096 cores in JuRoPA.

| | | Message Size | | | | |
|---|---|---|---|---|---|---|
| | | 256B | 4KB | 64KB | 1MB | 16MB |
| FT | Pull | 665.2 | 9706 | 63465 | 48409 | 48647 |
| | Pull-s-f | 650.5 | 9865 | 84944 | 162280 | 167094 |
| SD | Pull | 172.2 | 2501 | 7284 | 7764 | 10750 |
| | Pull-s-f | 172.6 | 2610 | 13003 | 26265 | 24031 |
| SVG | Pull | 11.86 | 188.0 | 1900 | 4094 | 4096 |
| | Pull-s-f | 11.83 | 184.5 | 1824 | 3194 | 3322 |
| JUDGE | Pull | 1550 | 22493 | 124537 | 108300 | 83310 |
| | Pull-s-f | 1521 | 22881 | 157871 | 248346 | 270849 |
| JuRoPA | Pull | 4369 | 68478 | 476794 | 320855 | 275361 |
| | Pull-s-f | 4387 | 66841 | 582289 | 1155182 | 1222592 |

## 5.2. Scalability and Performance of UPC Broadcast

Figure 5.6 shows the performance obtained for the different implementations of the broadcast operation in the Finis Terrae supercomputer. The most relevant algorithms for this system are the variations of the pull approach with static fragmentation, both with flat tree and binomial trees in the intra NUMA level, PGASCol (pull, stat. frag., flat) and PGASCol (pull, stat. frag.), respectively. The proposed algorithms present the highest benefit, both for 16KB and 1MB messages, on 16 cores, and also for 512 and 1024 cores. The efficient handling of intranode transfers is key for achieving a good 16-core performance result, whereas the scalable design on 512 and 1024 cores is key to outperform BerkeleyCol, which suffer performance drops for the largest core counts. 16KB message performance (top graphs) is dominated by start-up latency and synchronization whereas 1MB message performance is dominated by the ability to overlap communications and harness data locality. Furthermore, scalable algorithms do not show performance degradation as the number of cores increases. In fact, the proposed PGASCol algorithms scale almost linearly, whereas BerkeleyCol suffers from poor scalability. Thus, the bandwidth obtained by PGASCol (pull, stat. frag., flat) is more than 61 times the bandwidth of BerkeleyCol on 1024 cores.

Figure 5.7 presents the performance in the Superdome system. The best performer algorithms in this system are again variations of the pull approach using flat trees in the intra NUMA level, both with static and dynamic fragmentation. Here the OS scheduler has a major importance since it is a shared memory machine with 16 NUMA regions. Different core mappings might yield significantly different results. This is specially important in the medium message case, since it is latency bound rather than bandwidth bound. In this scenario, there is an added effect. UOMS implements a variability filter for small messages, that causes that the minimum reported time for a message size cannot be smaller than the minimum time for the size immediately smaller. The result is that a poor process distribution in the beginning of the benchmark causes the minimum reported time to be too high. This is why the PGASCol algorithms present more stable results for 16 KB messages, since the processes are optimally distributed from the beginning. However,



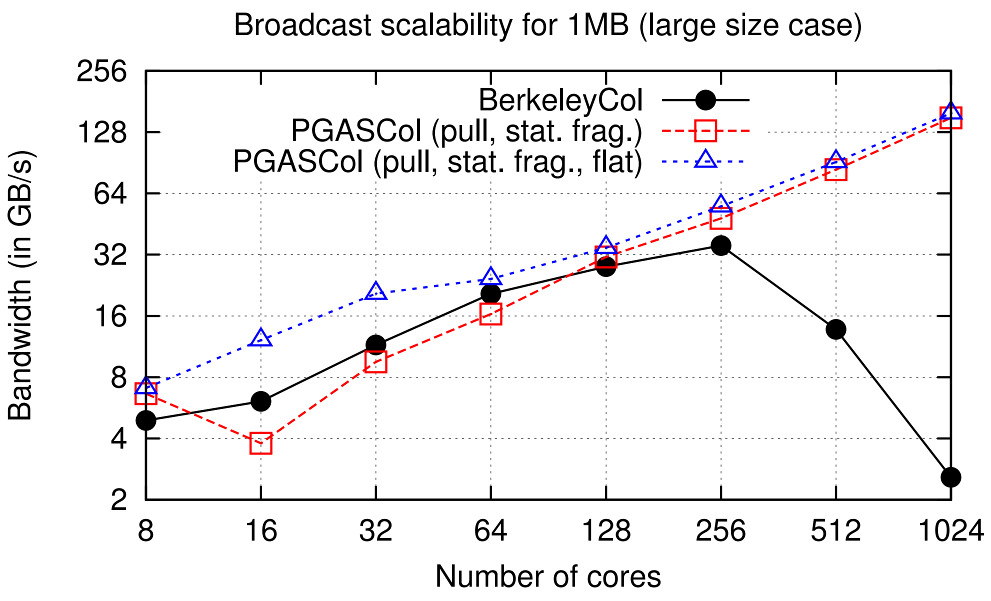
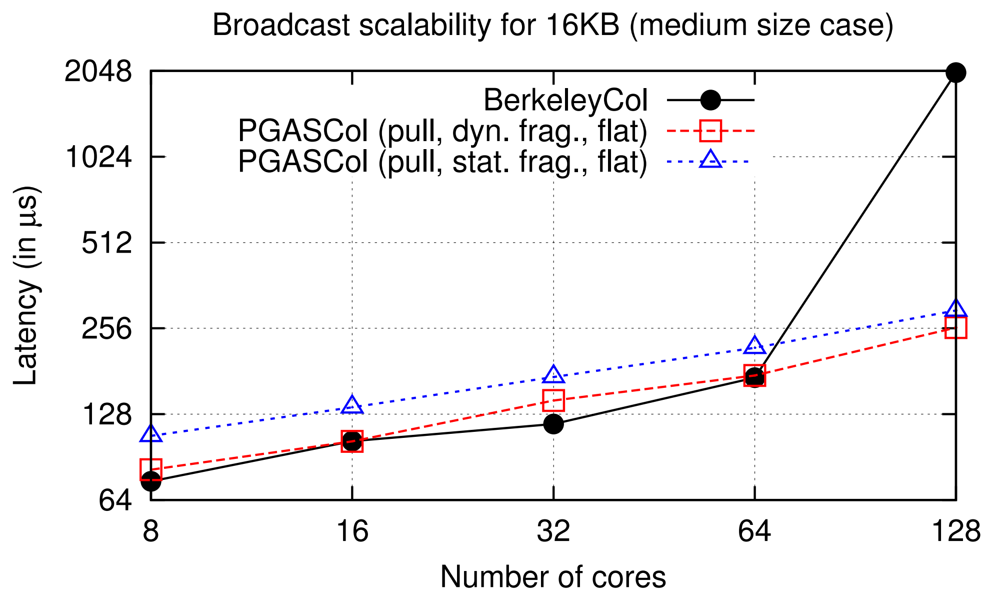


Figure 5.6: Broadcast performance and scalability on Finis Terrae

different runs can yield different results for BerkeleyCol, depending on the OS scheduler decisions, resulting in a non-predictable performance. This effect is not present in other systems or collectives –besides reduce–, as they are much less sensible to a bad process distribution, nor is it present in the averages reported. Regarding 1 MB communication scalability, PGASCol suffer when using 32 or 64 cores (2 or 4 processes per cell). However, its better scalability allows performance benefits of around 30% over BerkeleyCol.

The results for the SVG system can be observed in Figure 5.8. The best PGASCol performance results have been obtained with the pull version with flat trees in the intra NUMA level, and the pull version with dynamic fragmentation. The performance drops significantly for 16 KB messages when more than one node is in use. Additionally, there is an increasing difference between the PGASCol algorithms and BerkeleyCol when fully populating the first node. In systems using Magny-Cours processors or similar architectures with many NUMA regions, the NUMA awareness becomes more important. With more than one node the PGASCol algorithms are heavily penalized. The network is Gigabit Ethernet, with high latency. Since the PGASCol algorithms rely on semaphores for synchronizing, which are basically very short messages, they will suffer in latency bound scenarios, like the one depicted on the top plot. The bottom plot is bandwidth bound, and therefore the use of semaphores does not hurt the performance as much as in latency bound scenarios. The tree topology yields major gainings in this scenario. The pull approach with dynamic fragmentation performs almost 18 times better than BerkeleyCol, with 192 processes. However, despite its good performance compared with BerkeleyCol, the network prevents to achieve a good scaling. The bandwidth for 192 processes is just 28% higher than with 96 processes.

The performance numbers obtained in the JUDGE supercomputer are shown in Figure 5.9. This system shares some common features with the Finis Terrae supercomputer. Namely the network, even though JUDGE is equipped with a later generation of the InfiniBand standard (InfiniBand QDR 32 Gbps vs InfiniBand DDR 16 Gbps). The behavior of the different collective operations evaluated is also similar. The major difference is the gap between BerkeleyCol and the PGASCol algorithms in the medium message case, when a single node is in use. This gap is the result of the good latency of the QPI bus and the tree topology mapped to the



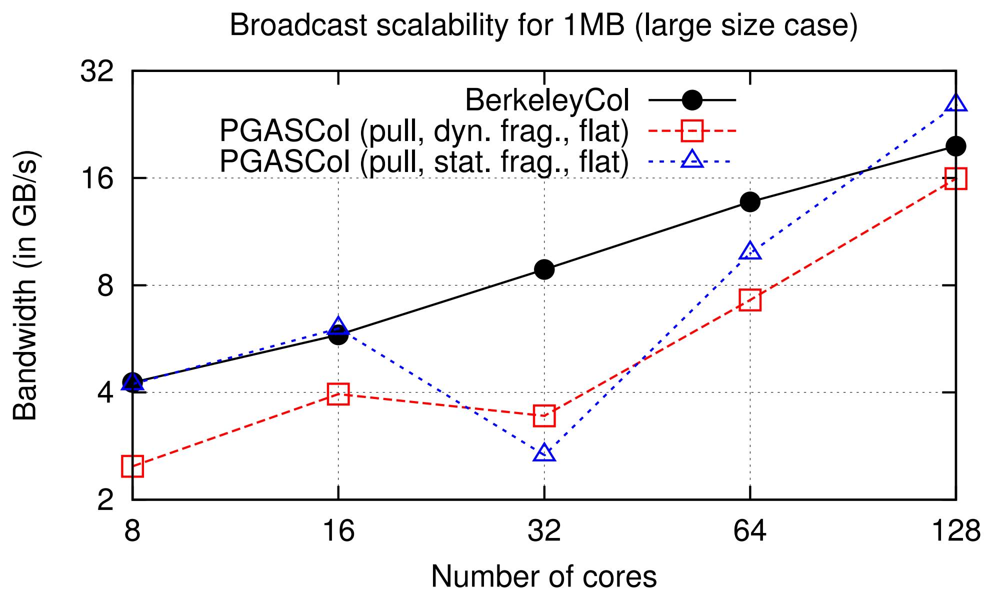
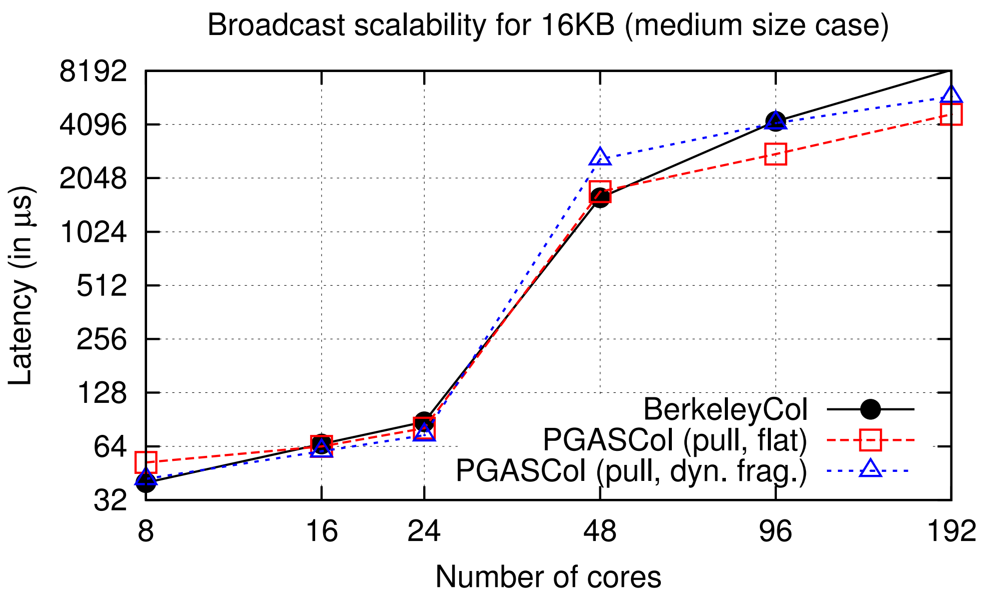


Figure 5.7: Broadcast performance and scalability on Superdome



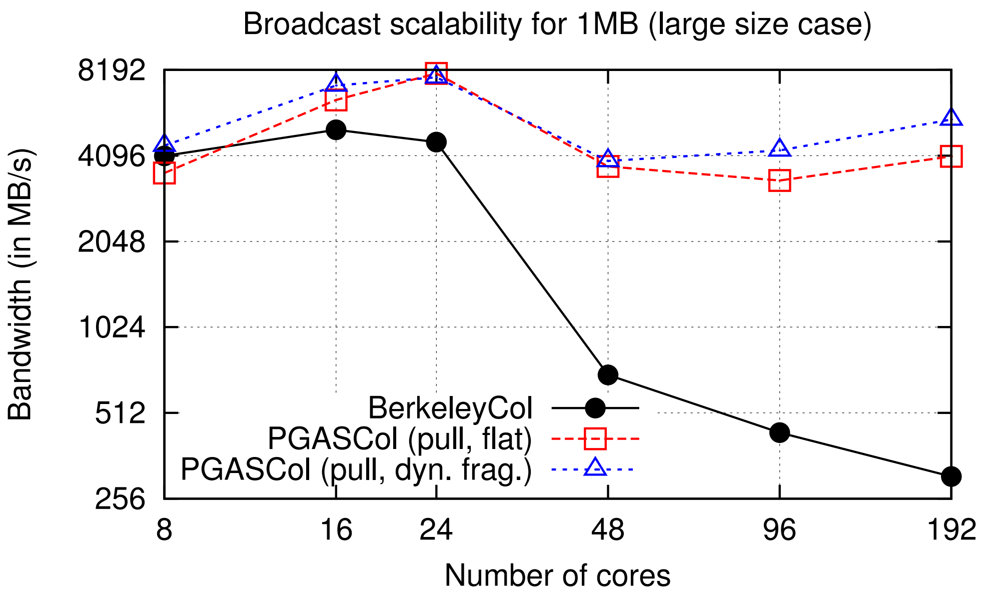


Figure 5.8: Broadcast performance and scalability on SVG

hardware, allowing the algorithm to achieve a good performance. The results for the large message case show the same tendency than in the Finis Terrae system. The BerkeleyCol collective does not scale beyond 192 processes, while the PGASCol algorithms keep scaling. For 648 processes the performance of the pull approach with static fragmentation is more than 420% of the performance of the BerkeleyCol collective.

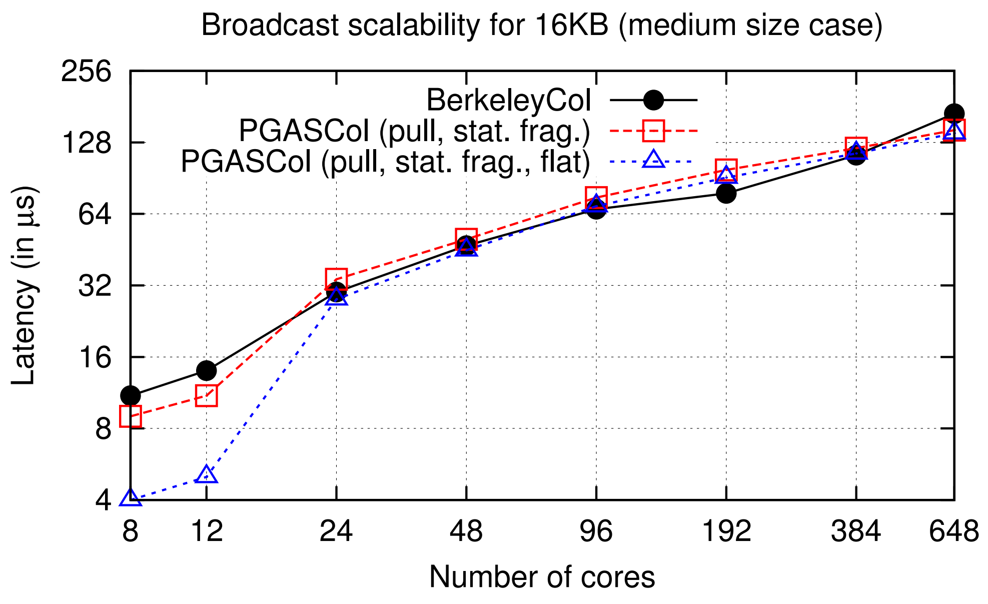
Lastly, the Figure 5.10 depicts the results for the JuRoPA supercomputer. This supercomputer also shares some architectural features with JUDGE and Finis Terrae, and the algorithms showed are the same as in these systems. The results are also similar to the results observed in JUDGE. However, there is one remarkable difference in the medium message case. Even though the performance of BerkeleyCol is slightly better than the PGASCol algorithms' performance in setups with a few nodes, with 1024 processes this gap disappears, showing the superior scalability of the PGASCol algorithms. In the large message case stands out the fact that the PGASCol algorithms keep scaling without hesitation up to 4096 processes. The PGASCol algorithm with pull approach and static fragmentation achieves almost 160 times the bandwidth obtained by BerkeleyCol, with 4096 processes.

The conclusions that can be extracted about this operation are: (1) the scalability of the developed algorithms is outstanding, especially for large messages; (2) for medium messages its performance is good within one node if the internal node buses are latency optimized; and (3) for medium messages BerkeleyCol usually performs better when more than one node is in use.

## 5.3. Scalability and Performance of UPC Reduce

There are no major differences between algorithms for the reduce operation. Therefore, all graphs will show the same two algorithms: Push and pull with flat trees in the intra NUMA level. The reduce operation is addition, and the data type double. The data size is per process. Therefore 2048 elements per process for 16KB message size, and 131072 for 1MB.

Figure 5.11 represents the performance for the reduce operation in the Finis Terrae supercomputer. The medium message case shows that the algorithms scale



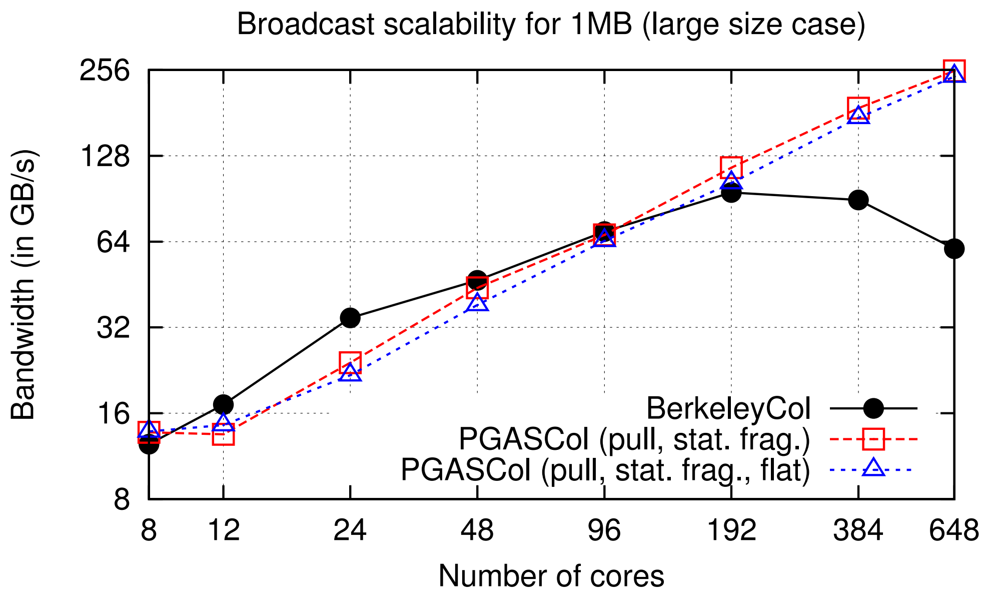
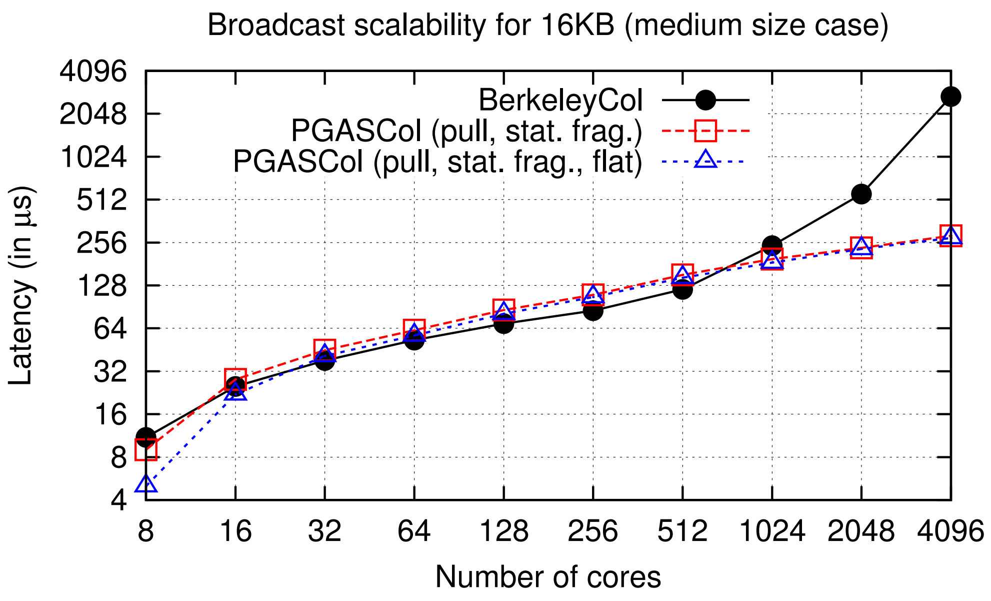


Figure 5.9: Broadcast performance and scalability on JUDGE



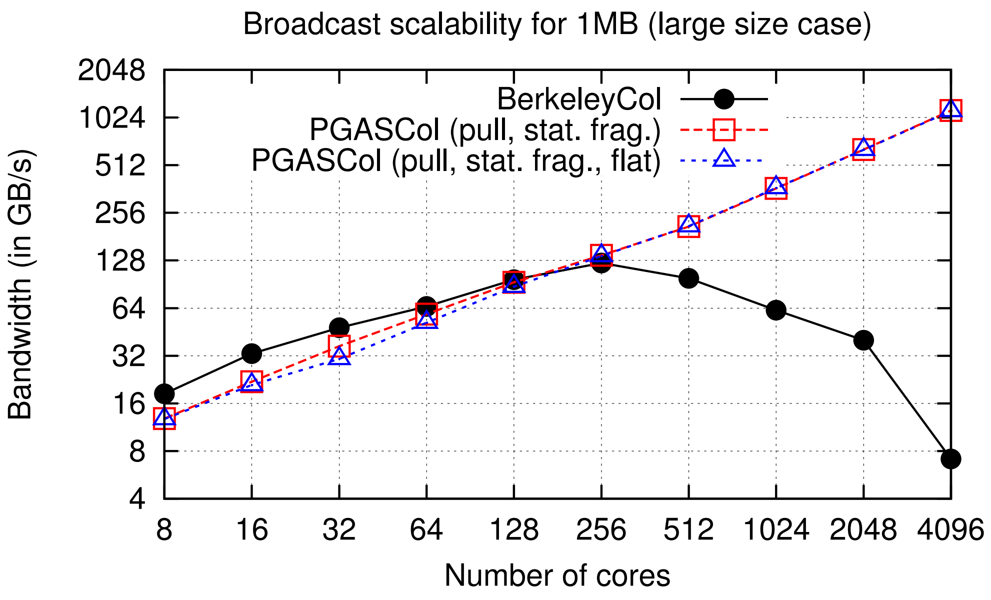
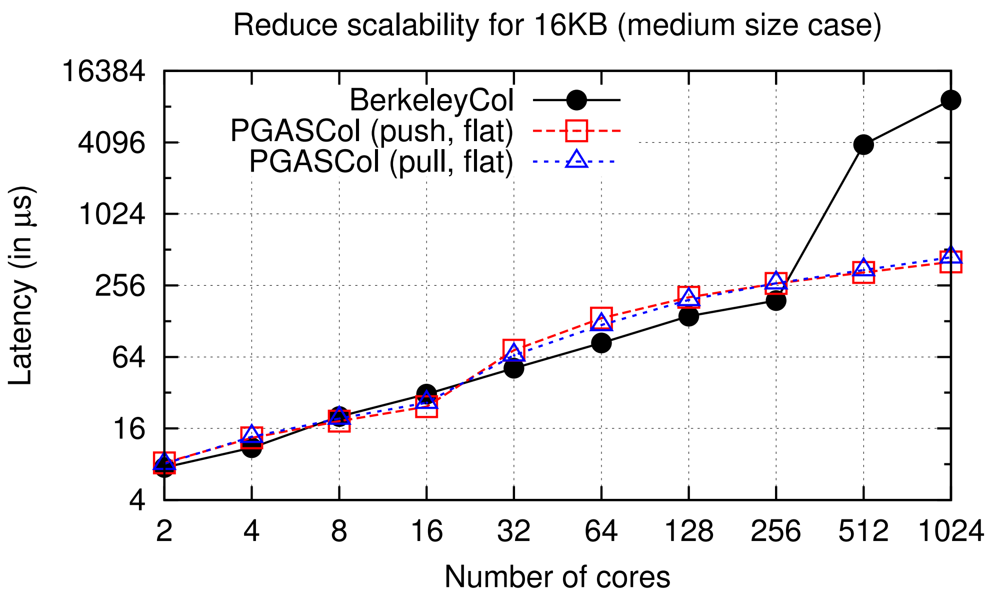


Figure 5.10: Broadcast performance and scalability on JuRoPA

steadily. This performance data is the most important one from the communication scalability point of view, since the large message case will be computational power bound. The data moved between processes will be the same, even though each process will have to spend more time computing the reduction of its own data in the large case. When using more than one node the performance is worse than BerkeleyCol by a narrow margin. Since BerkeleyCol has its collectives implemented in GASNet, rather than in UPC, its network access is slightly faster than directly from the UPC layer, causing this performance difference. Despite its slightly better performance, BerkeleyCol performs much worse for 512 processes or more. This fact is also present in the other InfiniBand systems. The issue can be attributed to the InfiniBand conduit or the reduce algorithm in BerkeleyCol. The large message case is not large enough on this system to be computational power bound. Therefore, when more than one node is in use the time to synchronize the processes is larger than the time spent computing, and top and bottom graphs are quite similar.

The Figure 5.12 shows the performance obtained in the Superdome system. In this shared memory system the results for the medium message case show that the BerkeleyCol reduce performs better than the PGASCol algorithms just with 2 processes. Up to 32 processes both algorithms performs at the same level. However, for 64 and, especially, 128 processes, the PGASCol algorithms keep scaling, while the performance of BerkeleyCol degrades, due to a poor process placement at the beginning of the benchmark and the variability filter of UOMS. This effect is observable for data sizes from 8 bytes to 16KB, not being present in the large message case. In the bottom graph the PGASCol algorithms outperform BerkeleyCol except for 4 processes. However, with the system fully populated the differences are not appreciable, since the limit is imposed by the caches and memory buses performance, and the operation is computational power bound.

The results for the SVG system are presented in Figure 5.13. In the medium message case the PGASCol algorithms outperform BerkeleyCol when just one node is in use (up to 24 cores). This is especially true when the node is fully populated (using 24 cores), due to the NUMA awareness. However, and as seen before, when more than one node is used, the PGASCol algorithms do not perform better than the BerkeleyCol counterpart. The large message case shows a scenario very similar to the medium message case when using more than one node, since the Gigabit



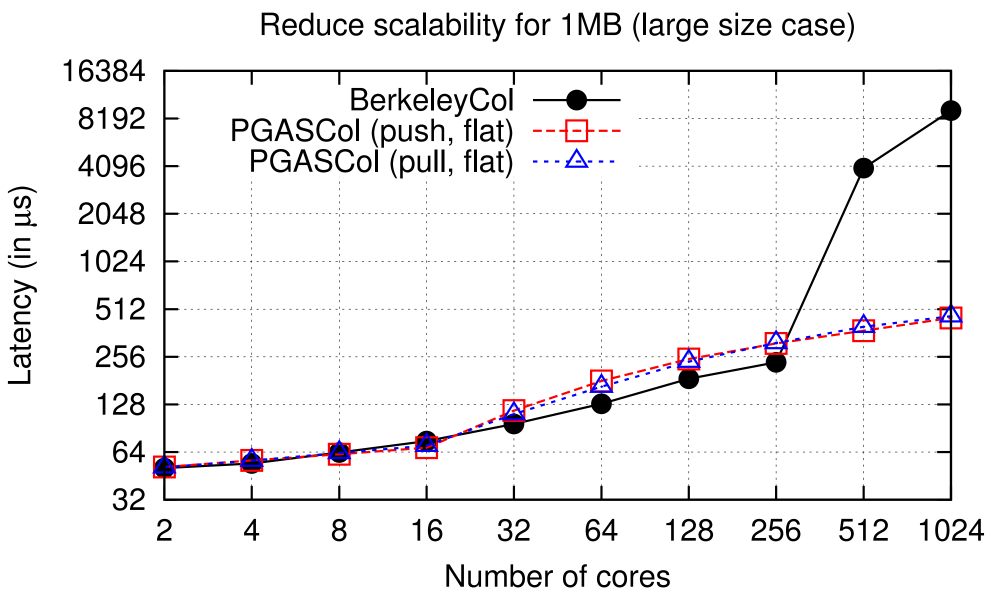
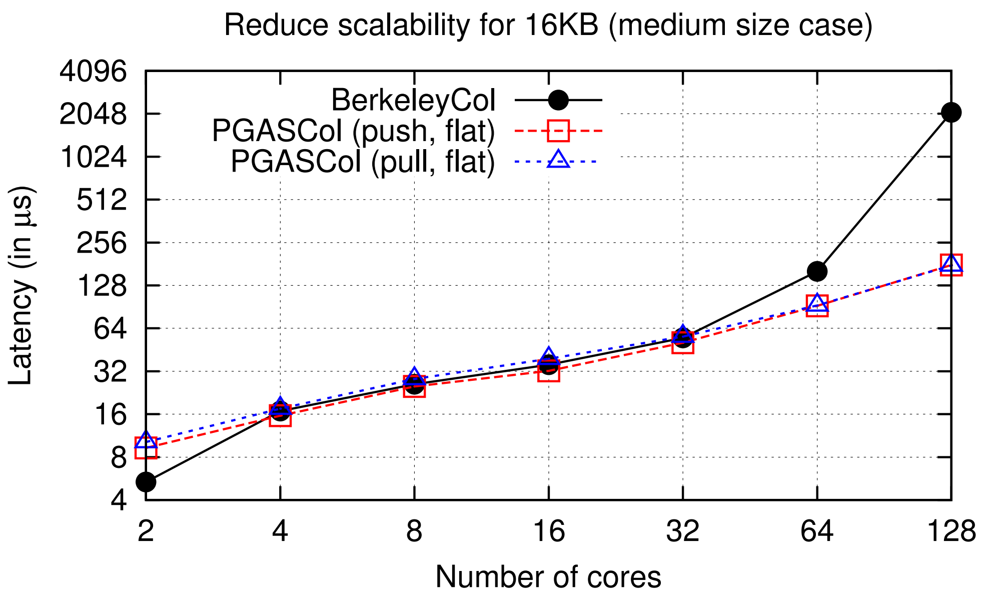


Figure 5.11: Reduce performance and scalability on Finis Terrae



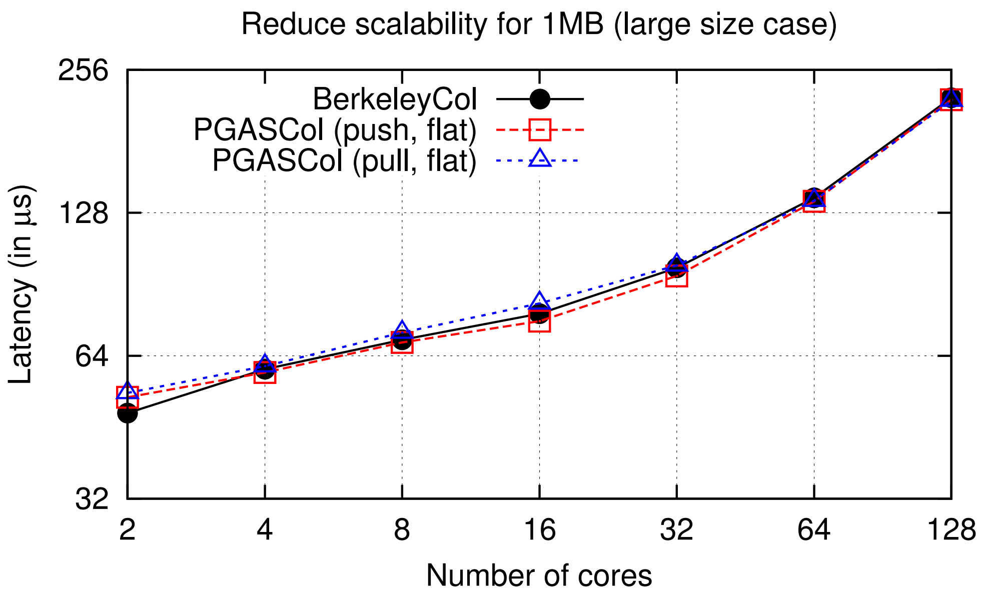
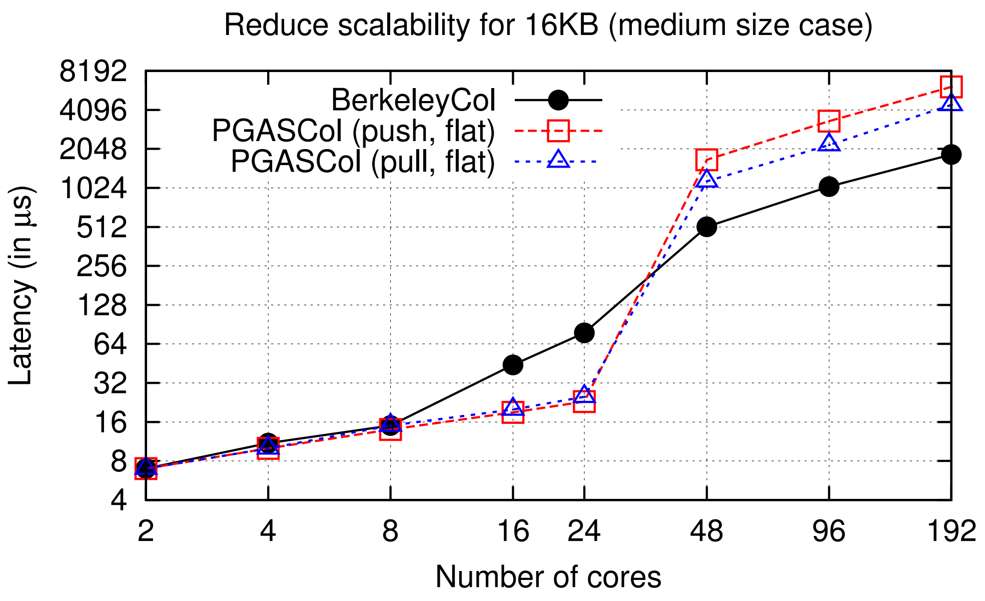


Figure 5.12: Reduce performance and scalability on Superdome

Ethernet interconnect becomes a major bottleneck and the benefits reducing the computational times are neglected by the high latency of the network (as it can be seen for the medium message case, the top graph). When using a single node BerkeleyCol and the PGASCol algorithms perform at the same level, except for the case with 24 processes. The reduce computational task is not optimized on this system, and therefore, as the data sets to be reduced get larger, the importance of the computing time increases, and the benefits of the PGASCol algorithms are neglected.

Figure 5.14 shows the results in JUDGE. The general shape of the plot in the medium message case is similar to the results for the Finis Terrae system. However, in JUDGE the PGASCol algorithms are able to better exploit the NUMA hardware than in the Finis Terrae. This is due to the fact that in JUDGE the caches are shared in the same NUMA region, since there is a single NUMA region per socket. However, in Finis Terrae there are 4 different processors per NUMA region. Therefore communication will be significantly faster in JUDGE between neighboring processes. In the Finis Terrae the ratio between speed communicating processes in different processors, but same NUMA region, and speed communicating processes in different NUMA regions is much lower. However, as for the previously analyzed systems, when more than a single node is being used, this advantage is lost due to the high network latency overhead which hides the differences between algorithms in the shared memory scenario. Regarding the 1MB performance results, BerkeleyCol is generally the best performer. This is because of the better implementation of the reduce computations in BerkeleyCol collective together with the fact that the QDR InfiniBand interconnection network is fast enough to make this setup computational power bound. Therefore, PGASCol collectives are only able to outperform BerkeleyCol when using 648 cores, as for more than 384 processes the performance of BerkeleyCol degrades sharply.

Lastly, Figure 5.15 displays the performance obtained in JuRoPA, a system which is similar to JUDGE in terms of architecture. The biggest difference, besides the size of the system, is the type of processors, with different number of cores (JUDGE has hexa-core Xeon Westmere processors whereas JuRoPA has quad-core Xeon Nehalem processors). Thus, in JuRoPA when using more than 8 cores (more than a single node), BerkeleyCol outperforms the PGASCol algorithms in both the medium and



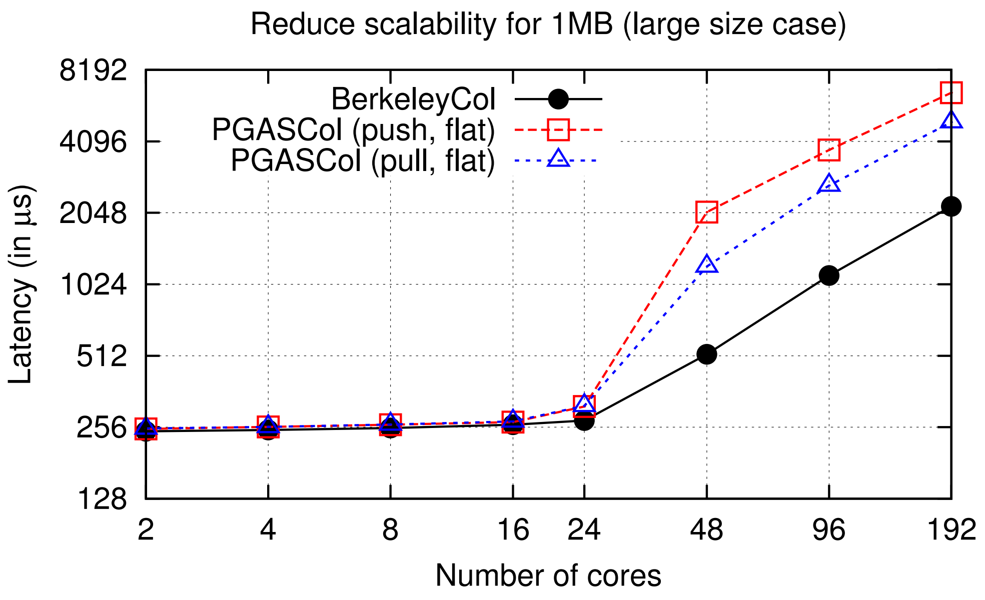
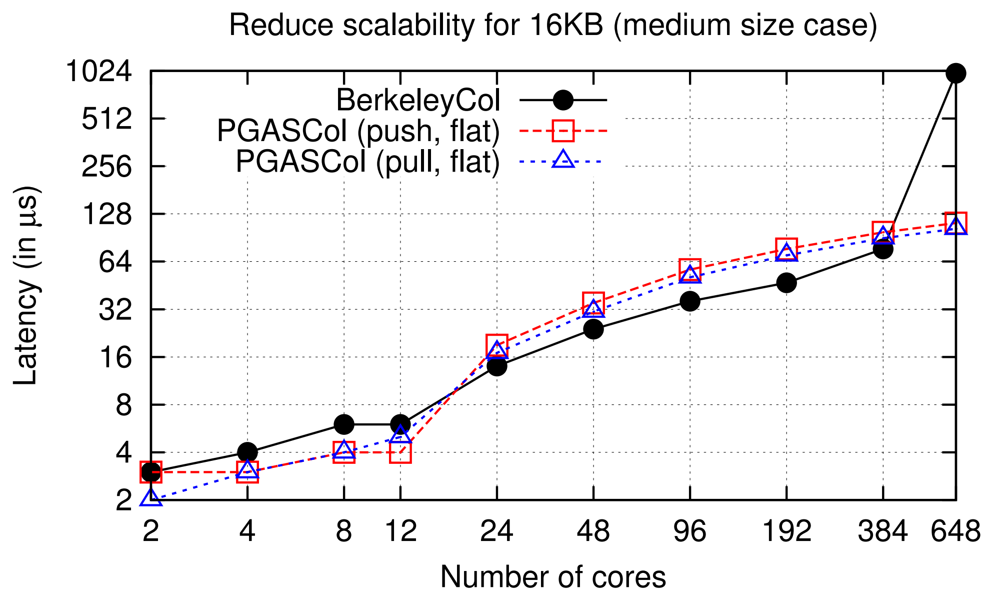


Figure 5.13: Reduce performance and scalability on SVG



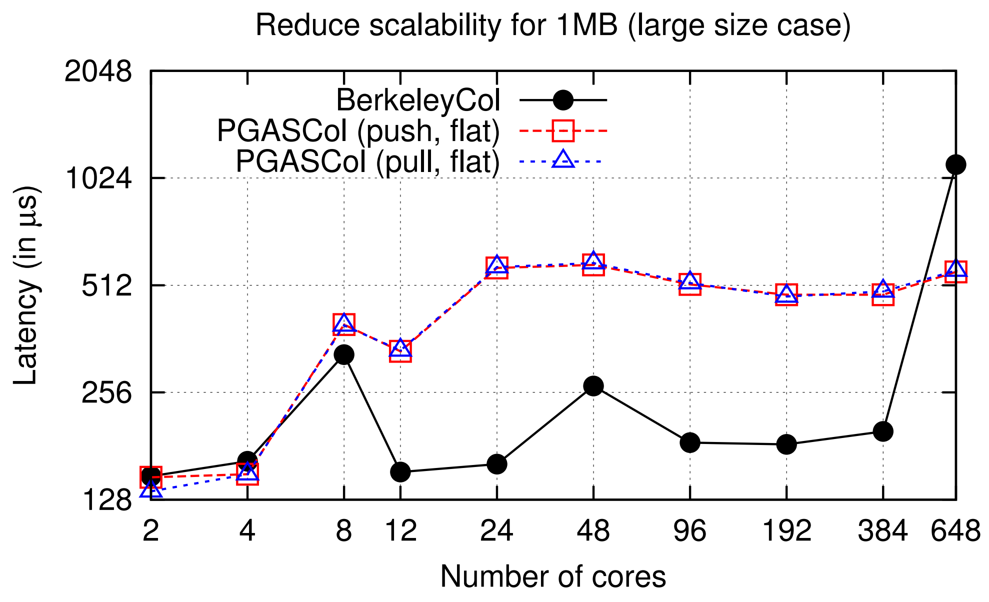


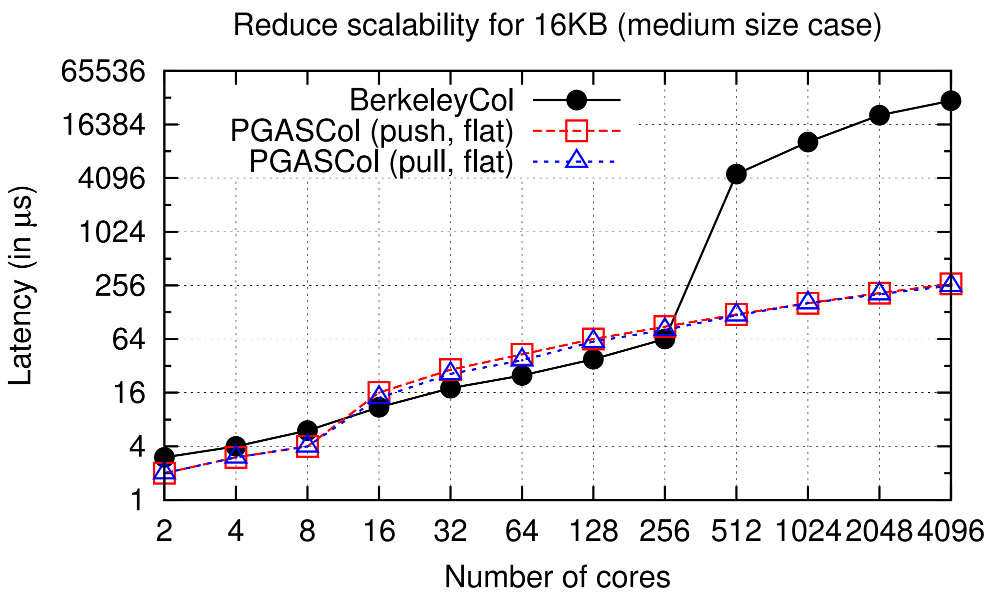
Figure 5.14: Reduce performance and scalability on JUDGE

the large message scenarios, but only up to 256 cores, as for 512 cores BerkeleyCol's performance degrades, whereas the PGASCol algorithms keep scaling steadily.

The analysis of the performance of the reduce implementations have allowed to draw the following conclusions: (1) the PGASCol algorithms can effectively outperform BerkeleyCol in modern NUMA hardware; (2) the performance of PGASCol reduce algorithms is latency sensitive, due to the synchronization and copy of single elements between processes, so therefore reducing network latency yields significant improvements, as observed when comparing systems with low latency networks (Finis Terrae, JUDGE or JuRoPA) with systems with high latency networks (SVG); and finally (3) BerkeleyCol presents a much more efficient implementation of the arithmetic operations supported in the reduce operation, which means that PGASCol reduce implementations have still room for improving its performance.

## 5.4. Scalability and Performance of UPC Scatter

The Michigan Tech University (MTU) reference implementation [70] of the scatter operation, unlike the broadcast and the reduce reference implementations (whose results were not shown for clarity purposes), presents a quite competitive performance despite its simplicity (it implements a flat tree). In the scatter operation the data from a root process has to be distributed (scattered) among all processes participating in the collective operation. The bottlenecks are, therefore, the outbound bandwidth of the root process and the start-up network latency. The simple algorithm implemented in the reference library is a good alternative due to that, since the use of the bandwidth of the root process is maximized without additional synchronization and copying overhead. As for previous collectives, for clarity purposes only the two best performer PGASCol algorithms are shown. Additionally, for scatter and gather the MTU reference library is considered and every graph will show its best performer algorithm (either the pull or the push version). Finally, it has to be noted that for scatter and gather the amount of data to be scattered/gathered increases with the number of cores. In this performance evaluation the selected message sizes are 16 KB an 64 KB. Therefore, by selecting 64 KB messages the root process will be handling 1 MB when communicating 16 cores (16 $\times$ 64 KB), or handling 256 MB when communicating 4096 cores (4096 $\times$ 64 KB).



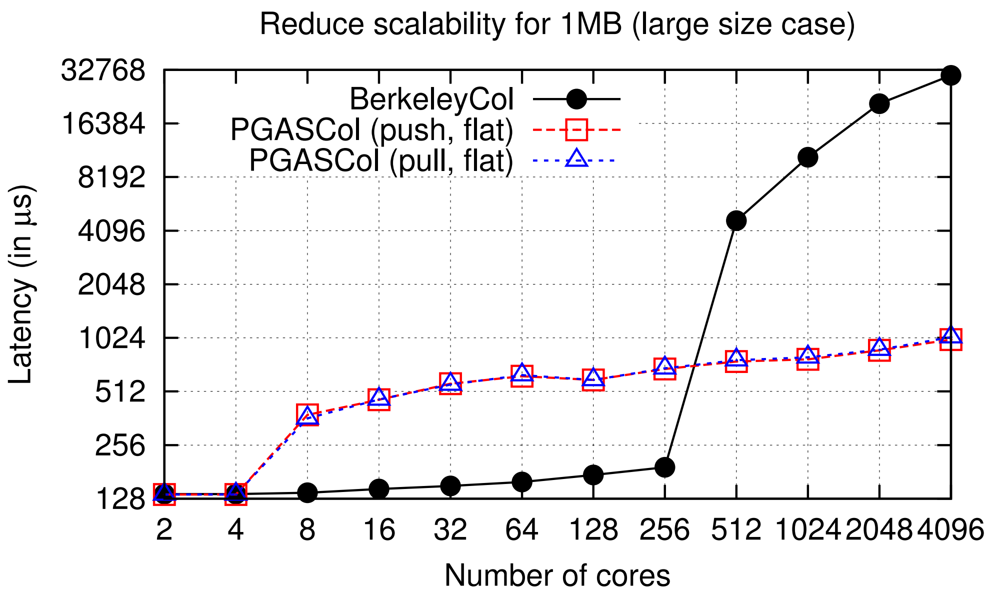
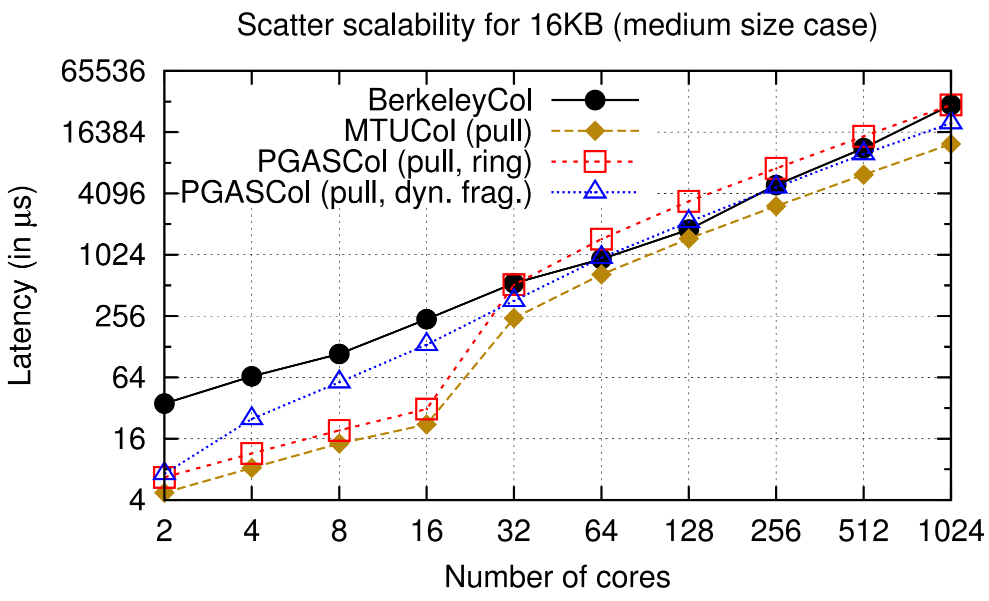


Figure 5.15: Reduce performance and scalability on JuRoPA

Figure 5.16 displays the results obtained from the benchmarking of the scatter operation in the Finis Terrae supercomputer. The relevant algorithms for this supercomputer are the pull versions of the MTUCol library, of the PGASCol ring algorithm and the PGASCol tree with dynamic fragmentation. In the 16KB case, in the one hand the best performer is the reference implementation, for the whole range of number of cores evaluated (2-1024). On the other hand, BerkeleyCol is the worst performer in shared memory (up to 16 cores), whereas it performs slightly better when using two or more nodes (from 32 cores), except when using 1024 cores (64 nodes). Here the ring algorithm presents the opposite behavior, as it performs well in shared memory (close to MTUCol performance), but it is the worst performer in the internode case. Finally, the pull version of the PGASCol tree with dynamic fragmentation has balanced performance, between the best and the worst case. The conclusions derived from the analysis of the performance results using 64 KB messages are different. Thus, BerkeleyCol is always the worst performer. Here MTUCol is the best performer, but in this case closely followed by the PGASCol ring algorithm. Once again the pull version of the PGASCol tree algorithm with dynamic fragmentation is not able to take advantages of its features because here the bottleneck is the outbound bandwidth of the root process. However, as for the 16 KB case, it presents performance results between MTUCol and the best performer PGASCol algorithm. When using 1024 processes the performance gap between the best performer and the worst performer is almost 1 GB/s, which in relative terms means that the best performer, MTUCol, presents 3 times higher performance than the worst performer, BerkeleyCol, which is not able to provide scalable bandwidth as the number of cores increases.

In Figure 5.17 the results measured in the Superdome system are showed. The best performer algorithms are the same as for the Finis Terrae, except for the PGASCol tree algorithm, which presents its optimal performance with static fragmentation. Here the differences between algorithms are much higher than in the Finis Terrae system, in both cases (16KB and 64KB). The reason is that the Superdome is a large NUMA server with lower communication latency than an interconnection network such as InfiniBand (the network in Finis Terrae). Moreover, in this shared memory system it is possible to access directly to the source data, minimizing problems such as congestion/contention like in a networked environment. Therefore, removing the interconnection network limitations (latency overhead, network con-



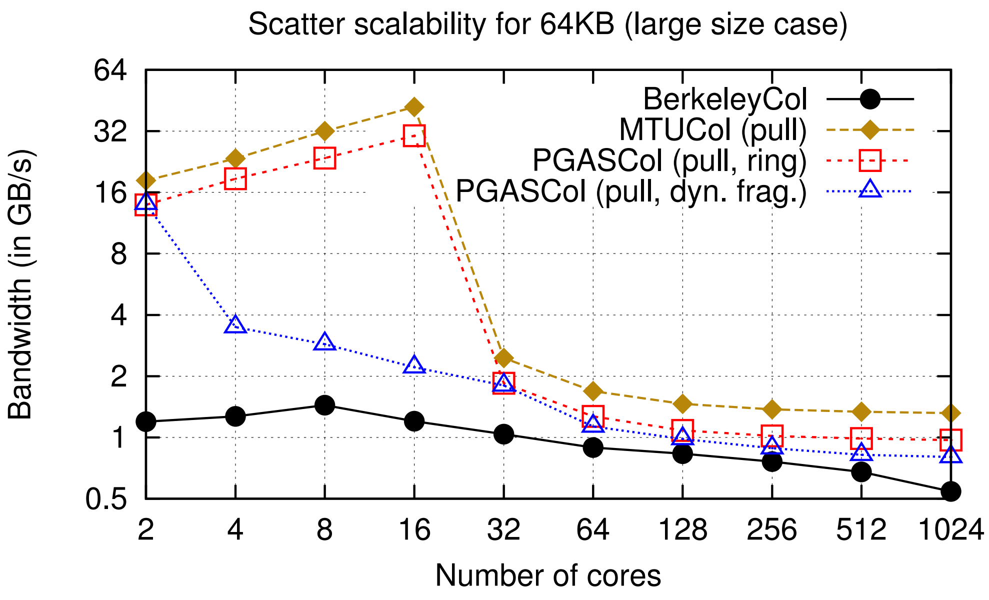
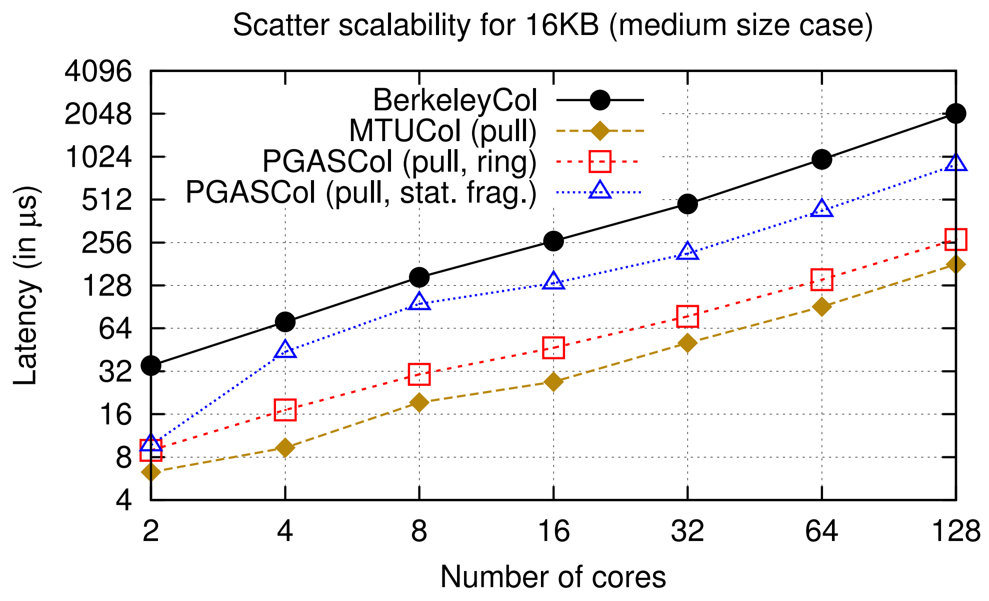


Figure 5.16: Scatter performance and scalability on Finis Terrae

gestion and contention) the differences between algorithms are more noticeable. In fact, the performance gap between the best (MTUCol) and the worst performer (BerkeleyCol) can be as high as 54 times, as for 64KB message size and 128 cores. As before, the best performer is the reference implementation, whereas the performance of BerkeleyCol falls behind all the other evaluated options, for 16KB and 64KB. The ring algorithm shows performance results around 30% lower than MTUCol, but following the same trend line, as both algorithms show very similar scalability. Finally, the PGASCol tree algorithm presents performance results quite close to the worst performer, BerkeleyCol, since the multiple levels the data has to go through, plus additional memory requirements and synchronizations, do not compensate. As can be derived from observing Figures from 5.16 to 5.20, the scatter operation is only able to scale on the Superdome, where BerkeleyCol is outperformed for 64KB messages on 128 cores by the PGASCol tree algorithm (2.75 times higher performance), the PGASCol ring algorithm (37 times higher performance) and MTUCol (54 times higher performance).

In the SVG the best performer algorithms are the same as for the Finis Terrae supercomputer, namely the pull versions of MTUCol, PGASCol ring and PGASCol tree with static fragmentation. The results can be seen in Figure 5.18. Here the results in shared memory (intranode, up to 24 cores) are similar to the Superdome system, although the MTUCol bandwidth is higher for 64 KB messages. However, for internode results (from 48 cores) the network latency overhead is a major issue, since the network available in this system –Gigabit Ethernet– presents a very high start-up latency. Thus, algorithms such as PGASCol ring especially suffers this high start-up overhead as it relies on semaphores, which are implemented using very short messages. Therefore, its performance on internode setups (from 48 cores) falls behind the remaining algorithms which are less sensitive to start-up network latency. BerkeleyCol, MTUCol and the PGASCol tree algorithm are able to take more advantage of the network, despite their limitations, avoiding synchronization overhead.

The results measured in JUDGE can be seen in Figure 5.19, where the best performer algorithms are the same as for the SVG and Finis Terrae. Moreover, the performance of these algorithms is quite similar to previous results on shared memory (intranode case, using up to 12 cores). However, when using multiple nodes



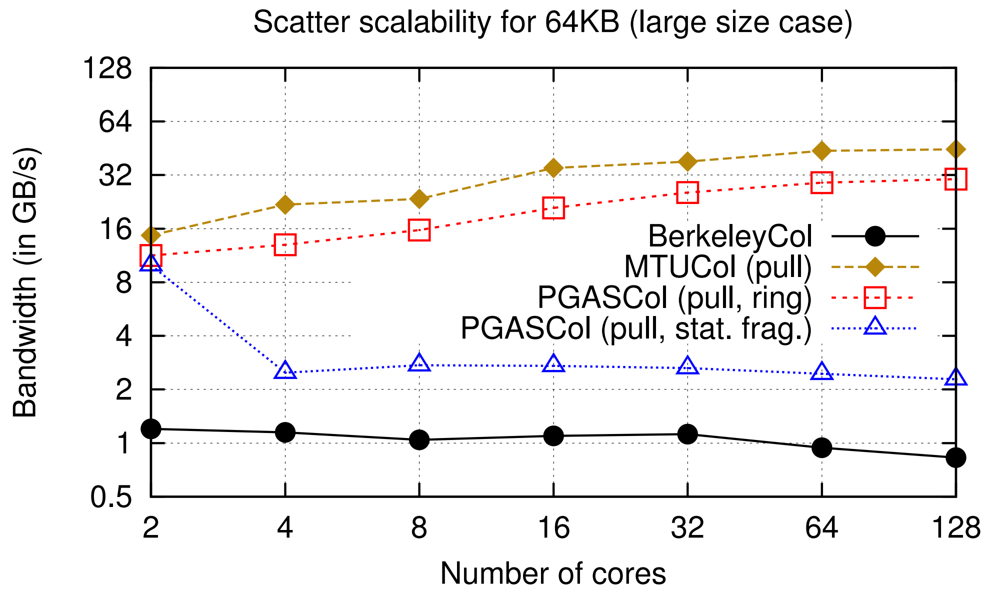
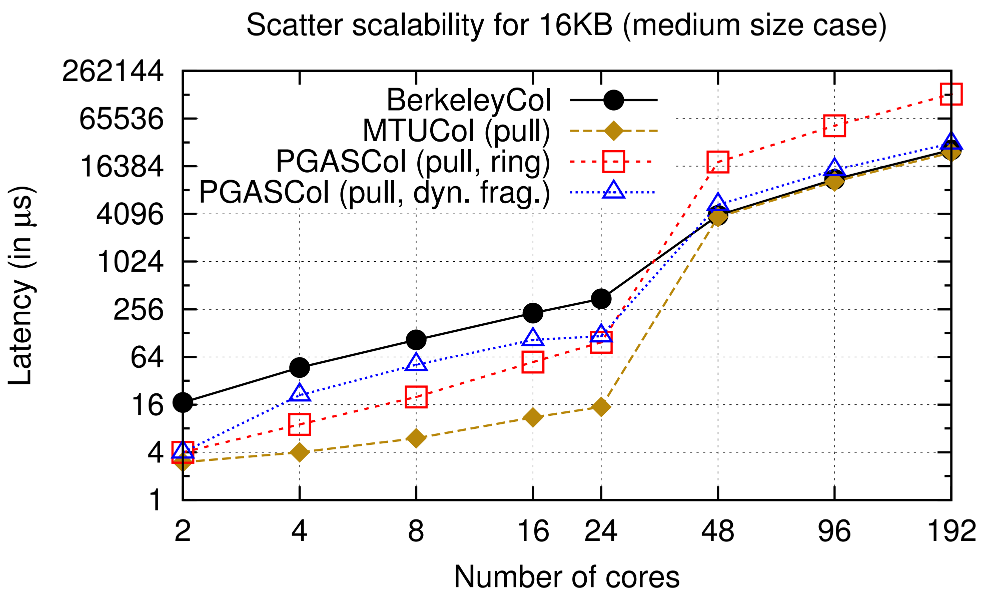


Figure 5.17: Scatter performance and scalability on Superdome



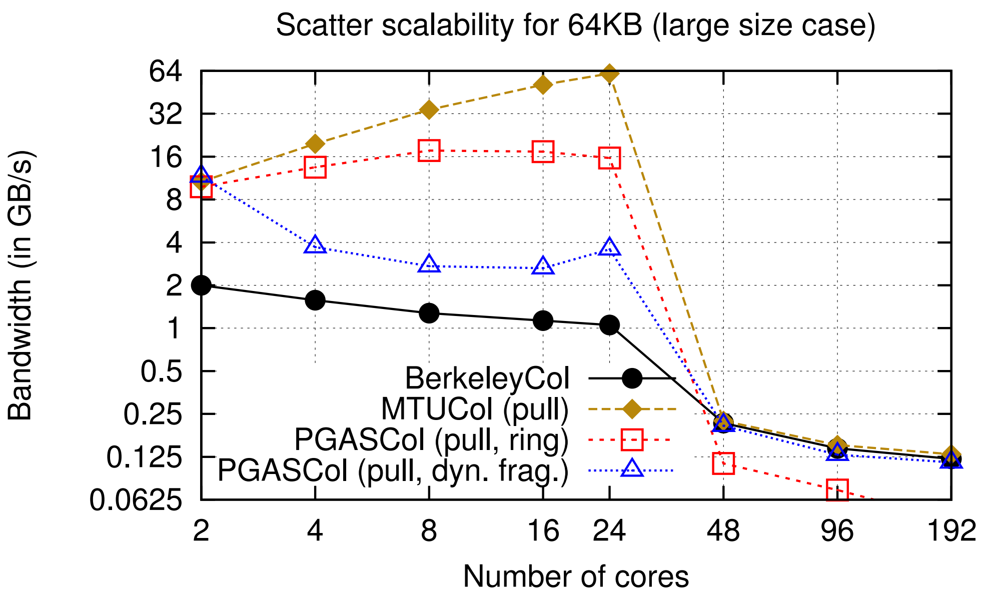
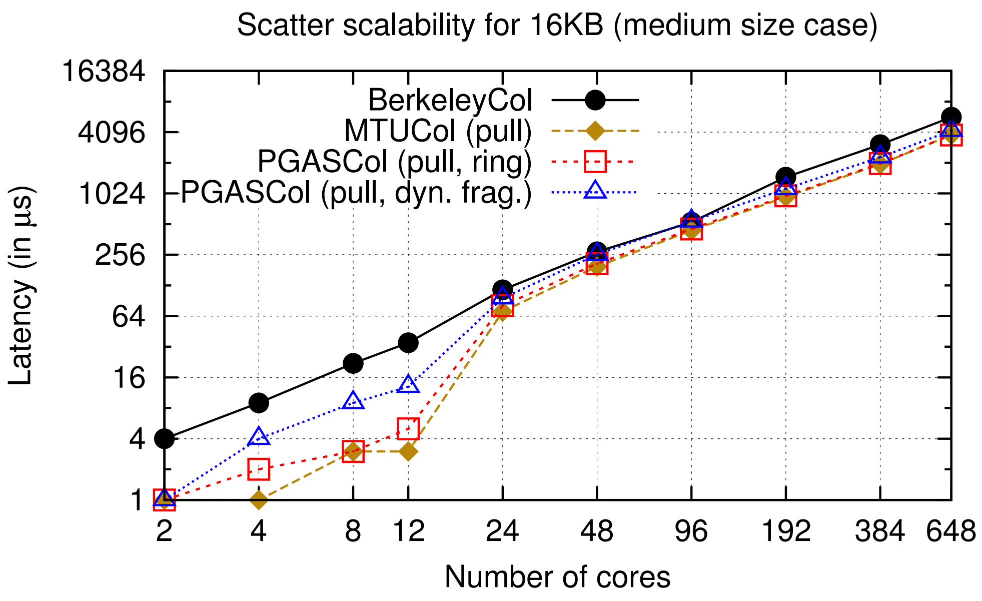


Figure 5.18: Scatter performance and scalability on SVG

–24 or more cores– the performance drops significantly, showing higher latency (top graph) or lower bandwidth (bottom graph). Thus, in this case the MTUCol and PGASCol ring algorithms show quite similar results, within 1% of performance gap for the 648 core setup, for both 16KB and 64KB. However, the use of PGASCol ring is recommended as communications are done one by one, coordinated by semaphores, which presents lower risk than the MTUCol implementation where all cores communicate to the root process, which could be potentially an important bottleneck. These two algorithms –the MTUCol and PGASCol ring– perform up to 60% better than BerkeleyCol on 648 cores.

Finally, the last system, JuRoPA, has an architecture similar to JUDGE, so it seems reasonable that the best performer algorithms are the same as for JUDGE, and that their performance results present similar behavior (they can be seen in Figure 5.20), so they share most of the analysis of the JUDGE results. Regarding JuRoPA benchmarking, the most important contribution is the analysis of the selected algorithms using up to 4096 cores. Thus, one of the conclusions of the analysis of the results is that MTUCol (pull), which implements a flat tree, is able to cope with up to 4096 simulatenous messages, even without degrading too much the performance, thanks to the InfiniBand network. However, BerkeleyCol can not avoid a significant performance drop for the 64KB test case using 4096 cores, falling in this case below half of the performance of MTUCol and PGASCol ring.

The conclusions that can be derived from the analysis of the performance results of the scatter operation are: (1) tree-based PGASCol algorithms, despite their scalability, are never the best option, due to the extra data that has to be handled; (2) the scatter operation is seriously limited by the outbound performance at the root process, which explains why quite simple algorithms, such as the flat tree implemented by MTUCol, are able to achieve the best performance although they might be disregarding the scalability of the data transfers; and finally (3) the MTUCol implementation has shown the best performance results and it has been able to deal with up to 4096 simultaneous communications, without saturating the interconnection network (in the evaluated system an InfiniBand network) and without requiring the implementation of any synchronization mechanism to support the scalability of the operation.



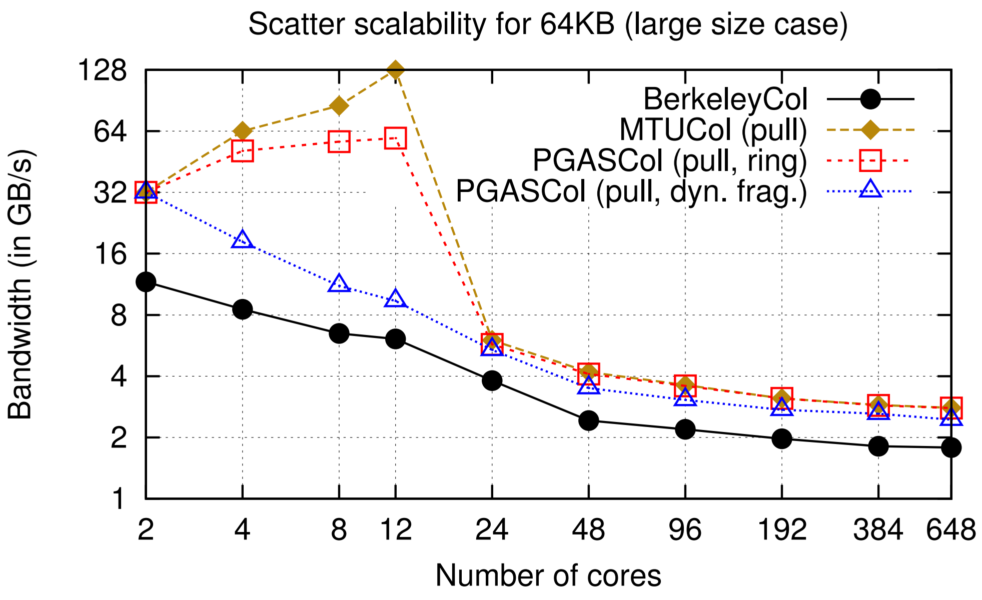
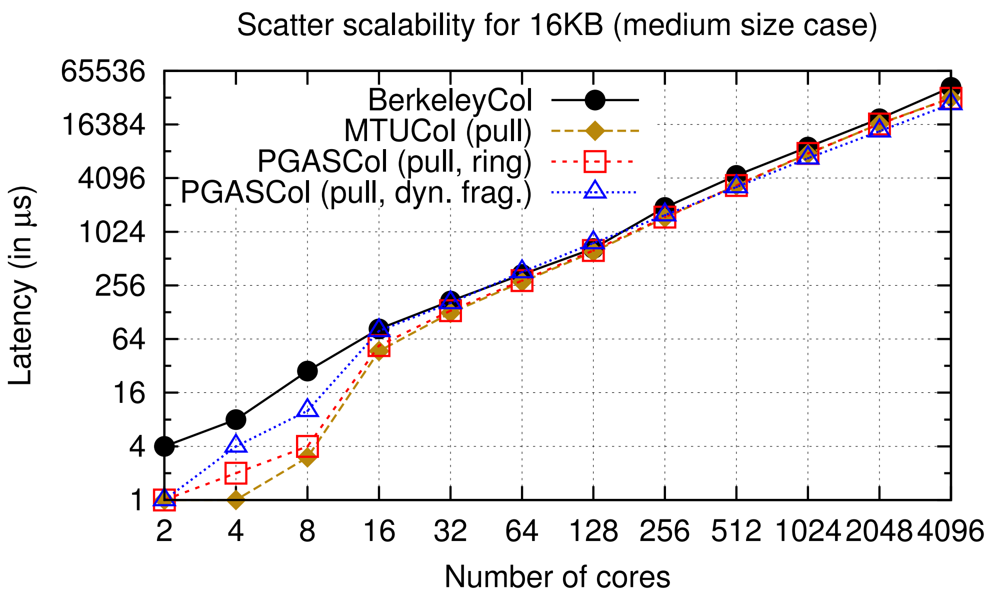


Figure 5.19: Scatter performance and scalability on JUDGE



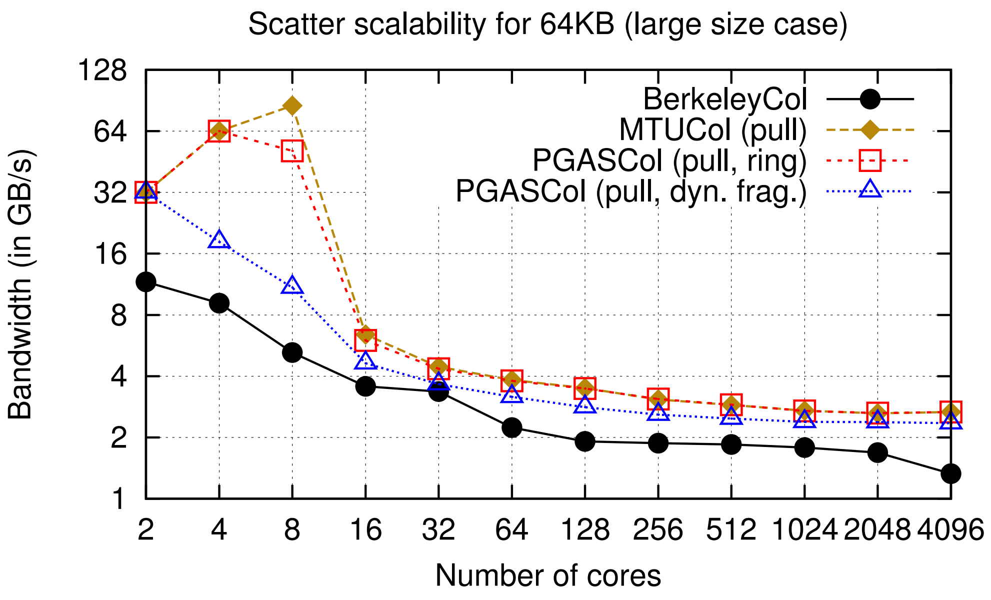


Figure 5.20: Scatter performance and scalability on JuRoPA

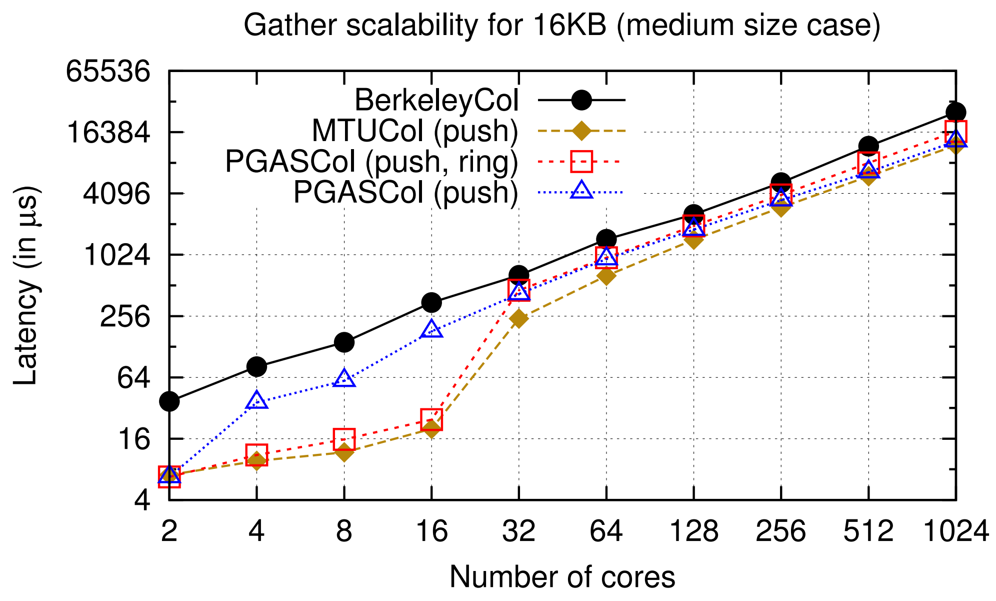
## 5.5. Scalability and Performance of UPC Gather

Figures from 5.21 to 5.25 present the performance results of the microbenchmarking of the gather operation on the 5 representative systems considered in this work. As for scatter, MTUCol, implementing a quite simple flat tree algorithm, has an outstanding performance. In this case the data is collected from all the processes and has to be written in the root process, so it is the reverse operation of the scatter and the analysis could be the same as for the scatter, just considering the reverse operation. Thus, the bottleneck is the inbound bandwidth and latency. Apart from the considerations about the direction of the communications, the gather operation presents performance results very similar to those of the scatter collective for all the systems and messages sizes. Thus, the analysis and conclusions for the scatter results are perfectly valid for gather. However, it shall be noted that whereas the best performer algorithms for scatter are those which implement pull-based approaches, for gather the best option is push. The reason behind that is that communications are initiated by all the participants, rather than just one. Therefore, the cost of setting up the communication is partly distributed, avoiding jitter and providing better overlapping.

## 5.6. Comparative Performance Analysis of NUMA Algorithms against MPI

This section presents a comparative evaluation of the proposed PGASCol algorithms against state-of-the-art collective algorithms, such as those available for MPI, which has been carried out in JuRoPA using the MPI implementation ParaStationMPI 5.0.27, based on MPICH2 1.4.1p1. Even though ParaStationMPI is not as widely spread as other MPICH2 derivatives, the fact that it is based on MPICH2 makes suitable for a reasonable comparison. Moreover, this is the MPI implementation installed and supported on JuRoPA, and therefore results with it are more significant for users of this system.

In order to allow comparisons as fair as possible, and due to the differences in how UOMS and IMB measure performance, described in Chapter 3, the reported



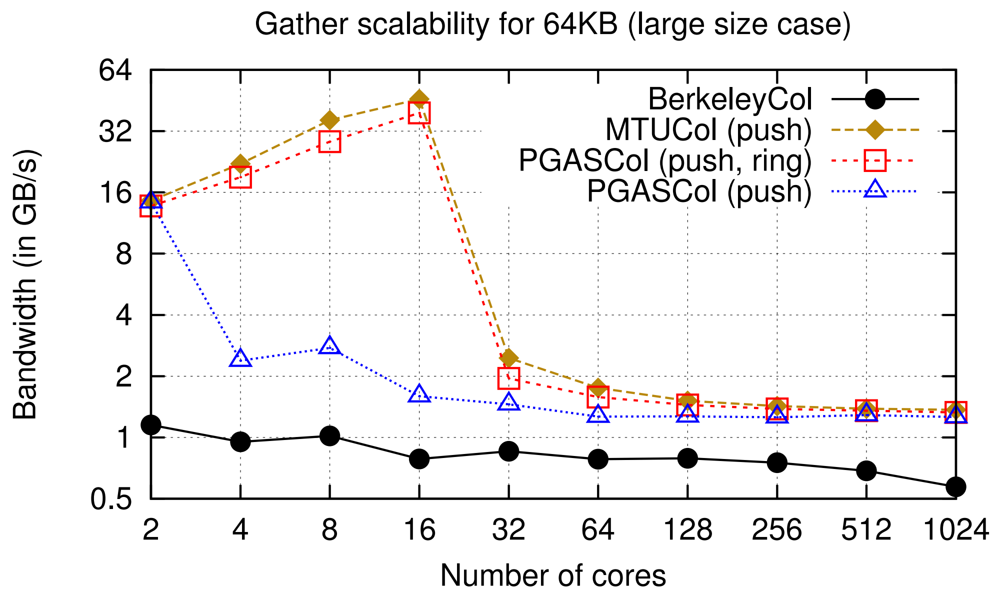
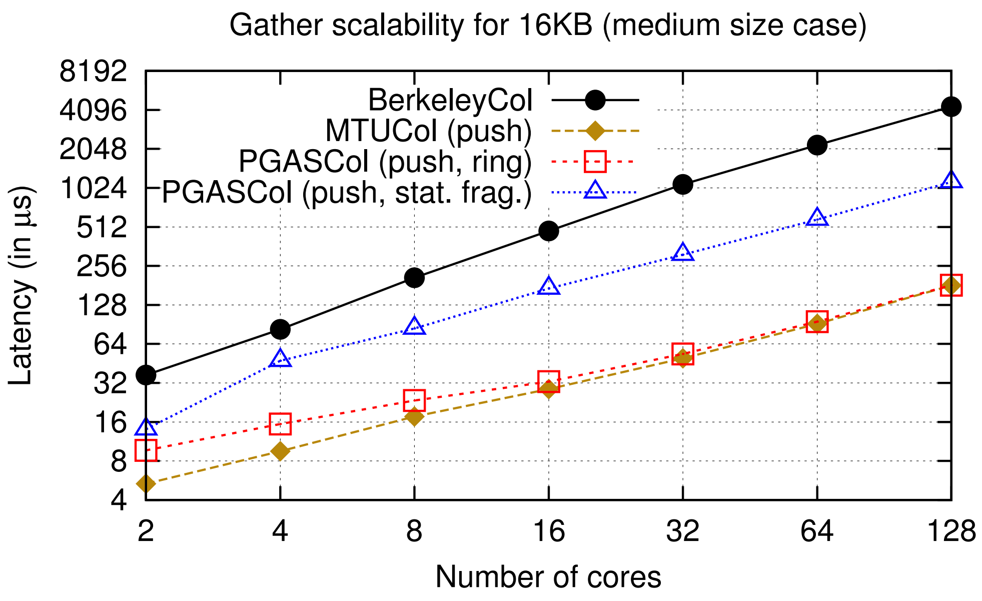


Figure 5.21: Gather performance and scalability on Finis Terrae



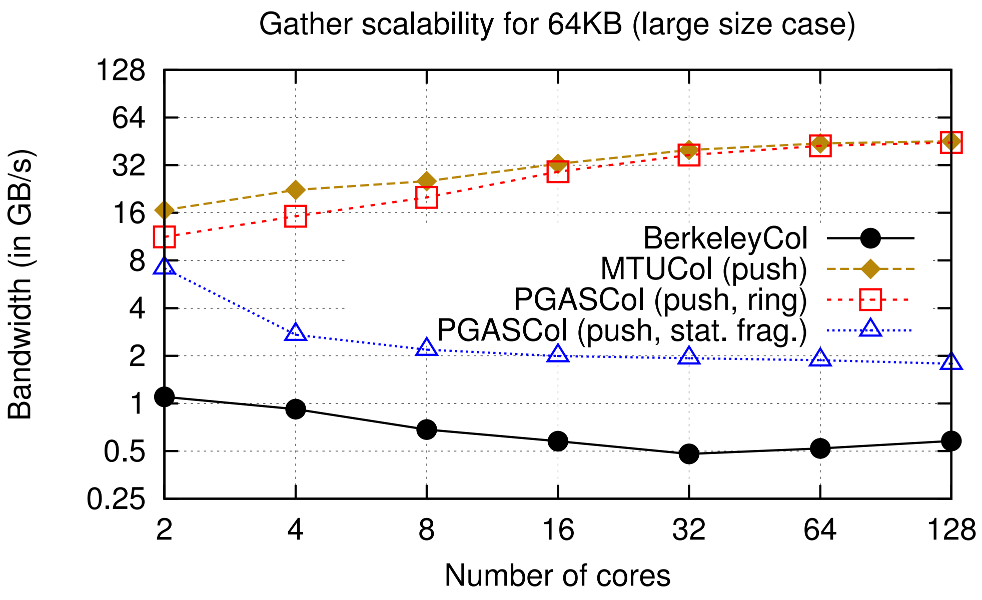
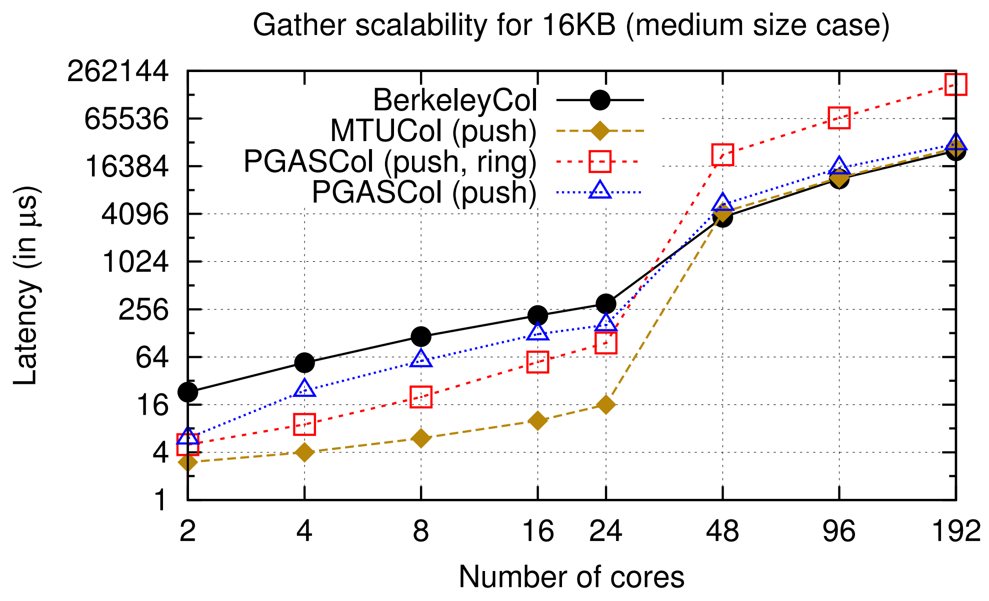


Figure 5.22: Gather performance and scalability on Superdome



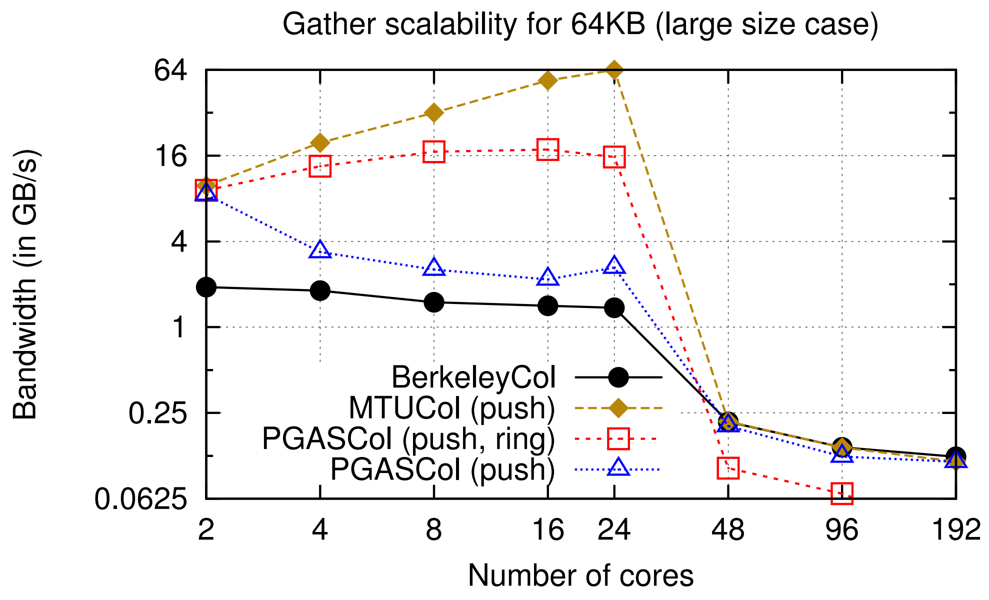
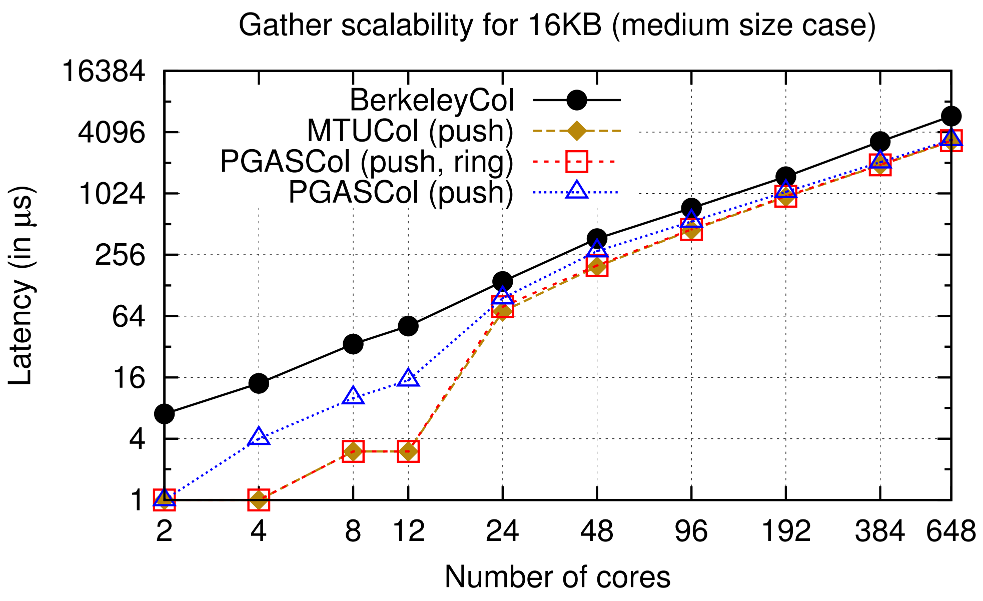


Figure 5.23: Gather performance and scalability on SVG



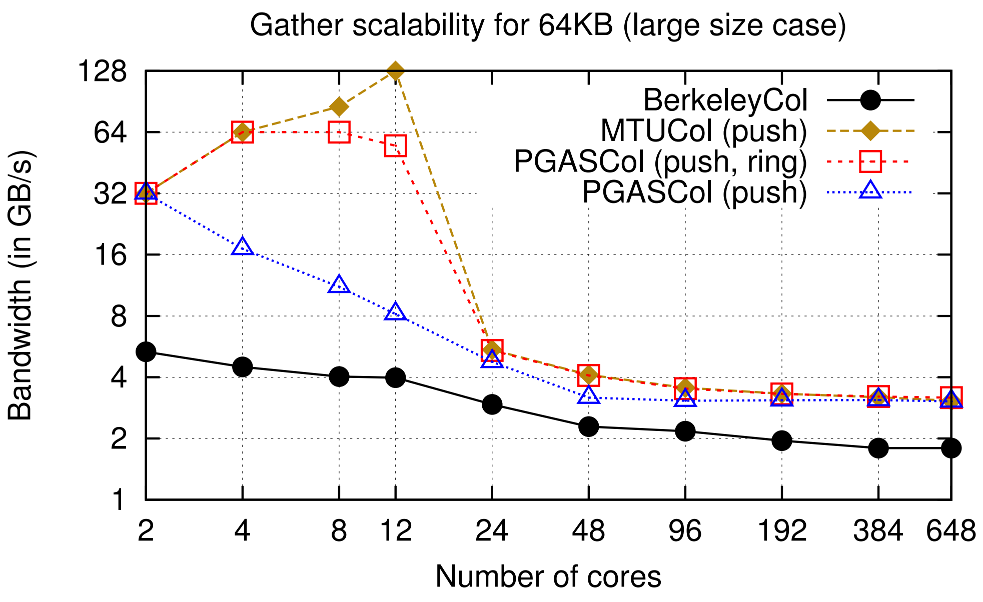
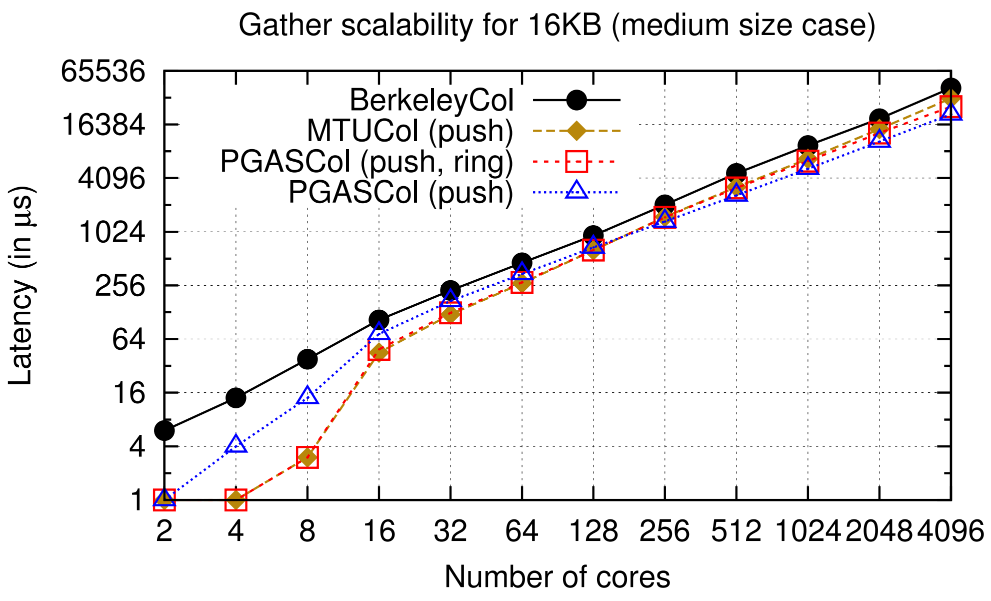


Figure 5.24: Gather performance and scalability on JUDGE



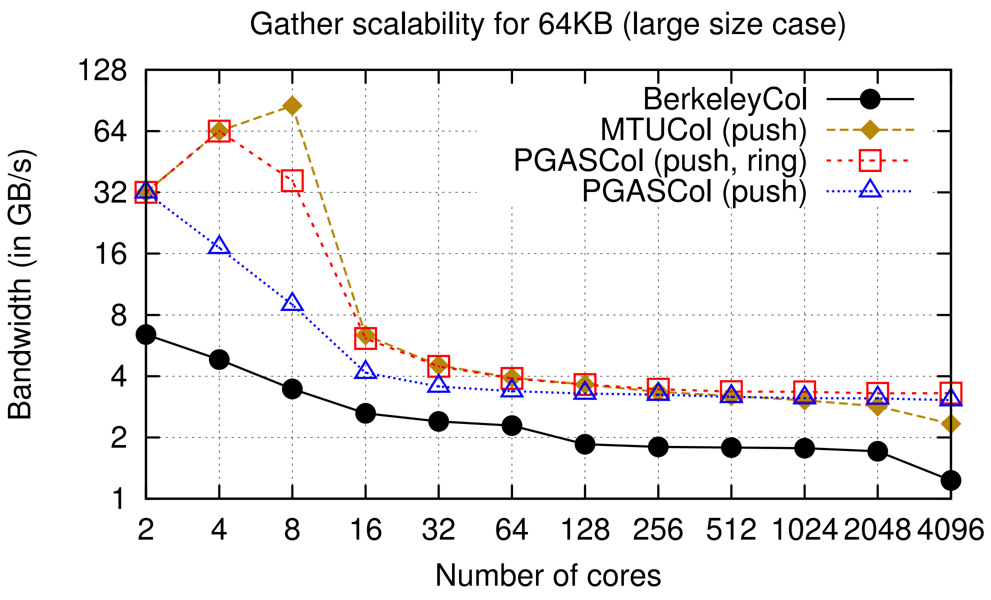


Figure 5.25: Gather performance and scalability on JuRoPA

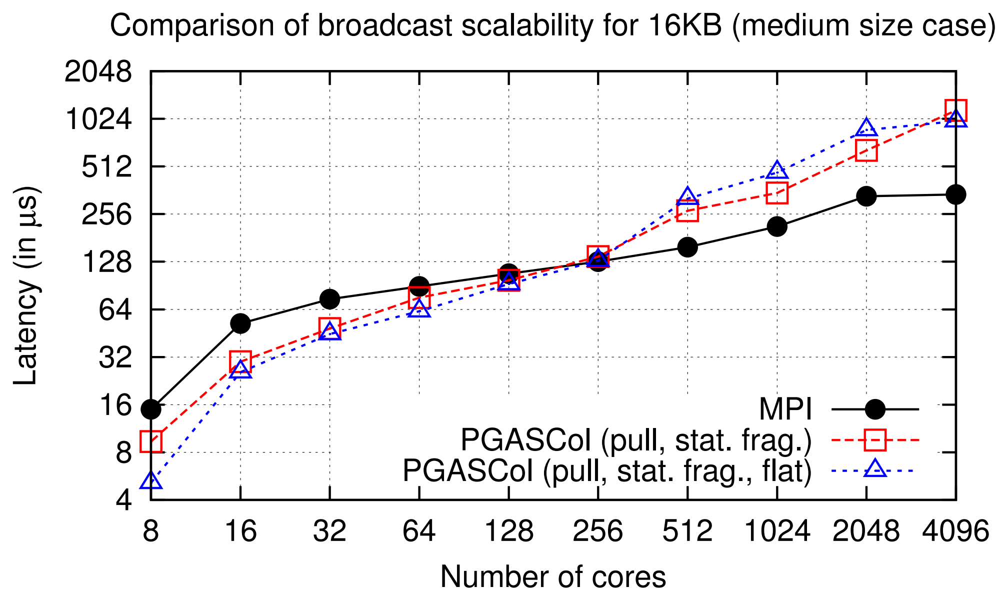
values for IMB are the maximum, i.e. the highest average time among processes, to guarantee a state where all the processes have finished the operation. The reported values for UOMS are the average, i.e. the average time per iteration needed to guarantee that all the processes have finished the operation. Both reflect the average time needed to allow the operation to be completed by all the processes. UOMS also reports the maximum bandwidth. Due to that, the reported bandwidth on this subsection is not the one reported from the output of UOMS, but the one calculated using the average latency.

MPICH2 implements three broadcast algorithms, selected at runtime depending on message size. These message size thresholds are configurable, but this evaluation uses the default thresholds. Thus, for messages up to 12KB the algorithm is based on binomial trees. For sizes between 12KB and 512KB the algorithm performs a scatter using a binomial tree and followed by an allgather implemented with a recursive doubling algorithm. For messages larger than 512KB the algorithm is similar to the previous one, except for the allgather phase, which is performed with a ring algorithm.

Regarding the scatter and gather operations, MPICH2 implements these collectives using an algorithm based on binomial trees, with intermediary buffers in non-leaf processes, in a similar way as the NUMA implementation proposed in this Thesis.

Finally, the reduce operation has been also included in this comparison. The reduce operation in UPC and MPI have significant differences. In UPC this collective is done on a shared array and produces a single value, whereas the outcome of the MPI reduce is an array result of reducing elements per position, using private arrays as source. However, when the number of elements per rank or UPC thread is 1, both operations are comparable. MPICH2 implements reduce using two algorithms: Rabenseifner's algorithm, for messages larger than 2KB, and a binomial algorithm for shorter messages. Since our comparison is limited to one element per rank, Rabenseifner's algorithm is not used.

Figure 5.26 presents the comparison of PGASCol with MPI for the broadcast using two representative message sizes, 16KB representative of medium size messages, and 1 MB, representative of large messages. For short messages (<12KB) the consid-



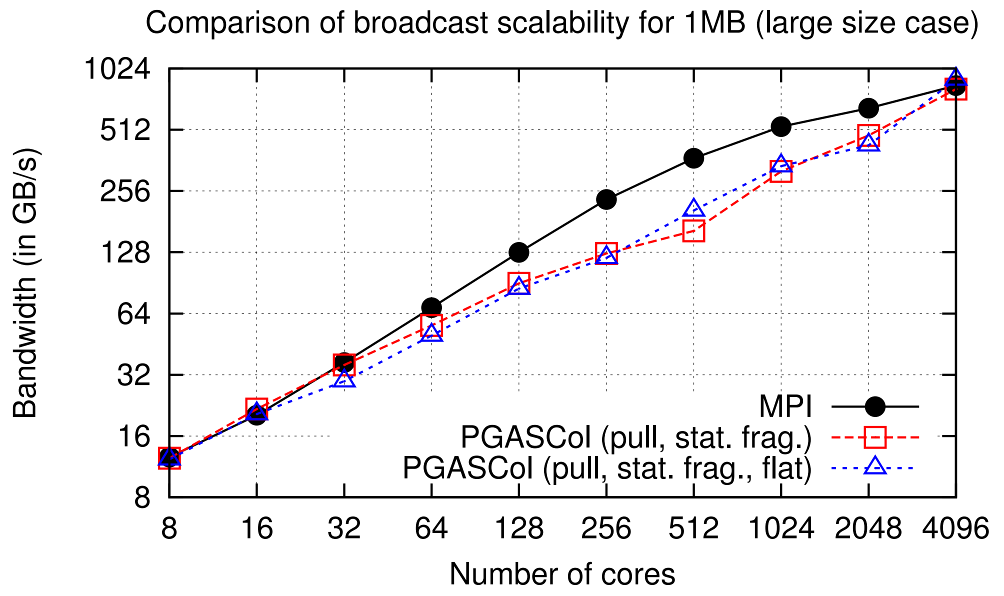
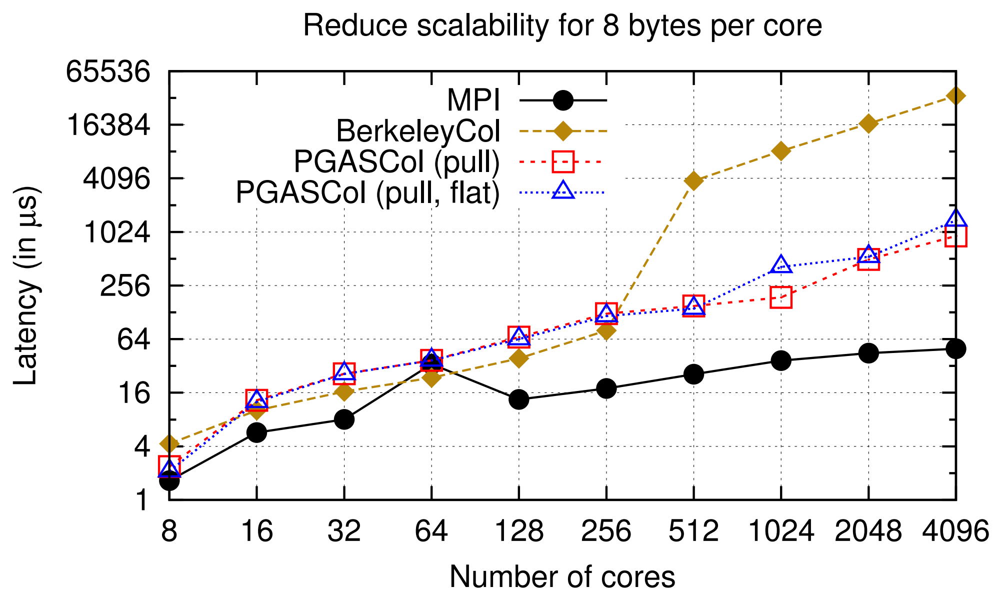


Figure 5.26: Comparison of broadcast scalability of NUMA-based algorithms against MPI on JuRoPA

ered algorithms have similar scalability, whereas the performance is highly dependent on the start-up latency achieved by MPI and UPC communications. Both graphs show the generally better performance and scalability of the PGASCol algorithms, although it is noticeable that MPI achieves the highest performance for 1MB from 64 up to 1024 cores. However, for 2048 and 4096 cores MPI performance is overcome by the better scalability of the PGASCol algorithms. These results demonstrate the significant benefits provided by the NUMA-based algorithms, which impact positively performance, in particular for medium messages and high core counts and large messages.

In Figure 5.27 the results for reduce can be observed. Both plots contain the data for 8 bytes (a double per MPI rank or UPC thread). The plot on the top represents latency, whereas the plot on the bottom represents MFLOP/s. In this range, with a message size of just 8 bytes, the best algorithms are both pulling algorithms. However, despite their good scalability, their performance is worse than for MPI, and in some cases worse than BerkeleyCol. With this setup, all the algorithms (except PGASCol with flat tree at the NUMA level) are algorithms based on binomial trees. The number of cores per node and per NUMA region is power of 2, and therefore the shape of the trees and the cost of the operation is the same between them. However, MPI outperforms all the UPC implementations, due to its lower start-up latency, that is specially important in this case due to the fact that this operation is largely dominated by the network latency. This fact is also the root cause for the low number of MFLOP/s, due to the low computation/communication ratio.

Figures 5.28 and 5.29 show the comparative performance results for scatter and gather, respectively, with a format similar to the layout previously presented. Thus, the selected message size for evaluation are 16KB and 64KB. Regarding the performance results, generally MPI is the worst performer and the PGASCol ring algorithm the best performer (up to 3 times more performance than MPI), especially for 64KB. MTUCol also outperforms MPI. Here the PGASCol tree algorithm is basically the same as the MPI algorithm. There are only two major differences: the NUMA affinity support, not present in MPI, and the fragmentation of the messages. These two differences explain the better performance of the PGASCol algorithms in scatter. For gather the fragmentation does not add any benefit and the best algorithm is the one that does not use fragmentation. Therefore, in this case the key



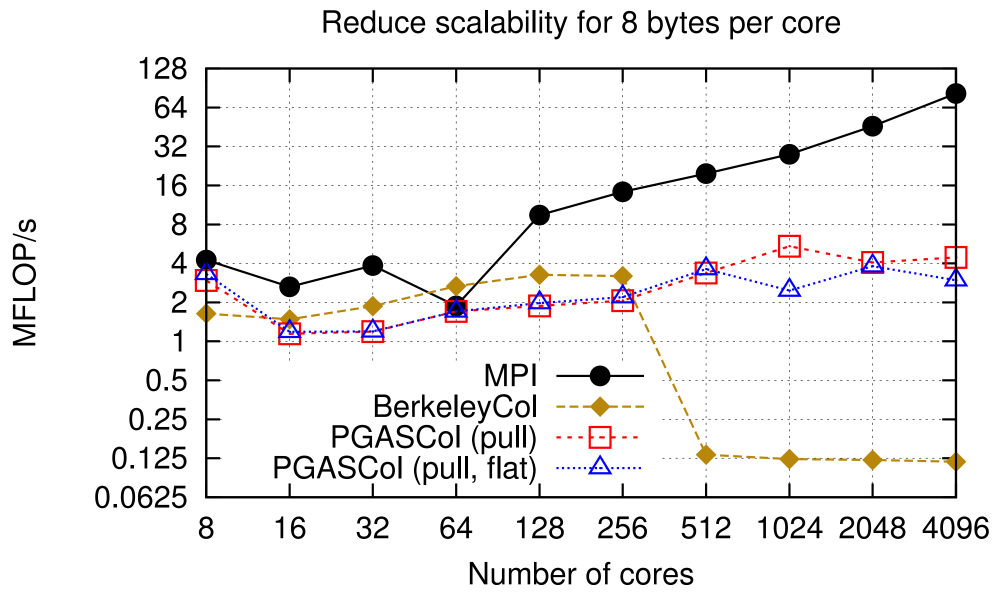
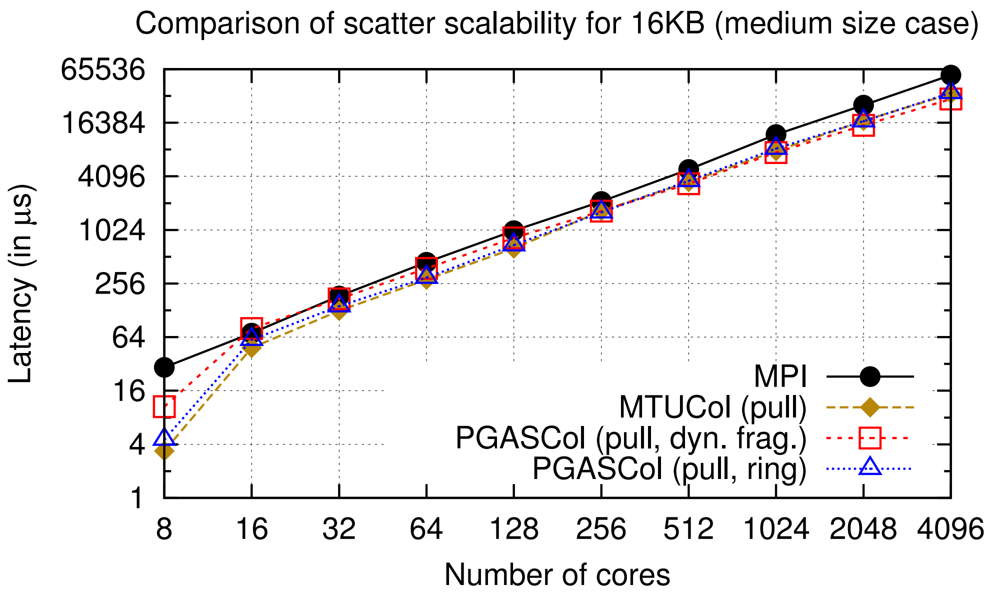


Figure 5.27: Comparison of reduce scalability of NUMA-based algorithms against MPI on JuRoPA



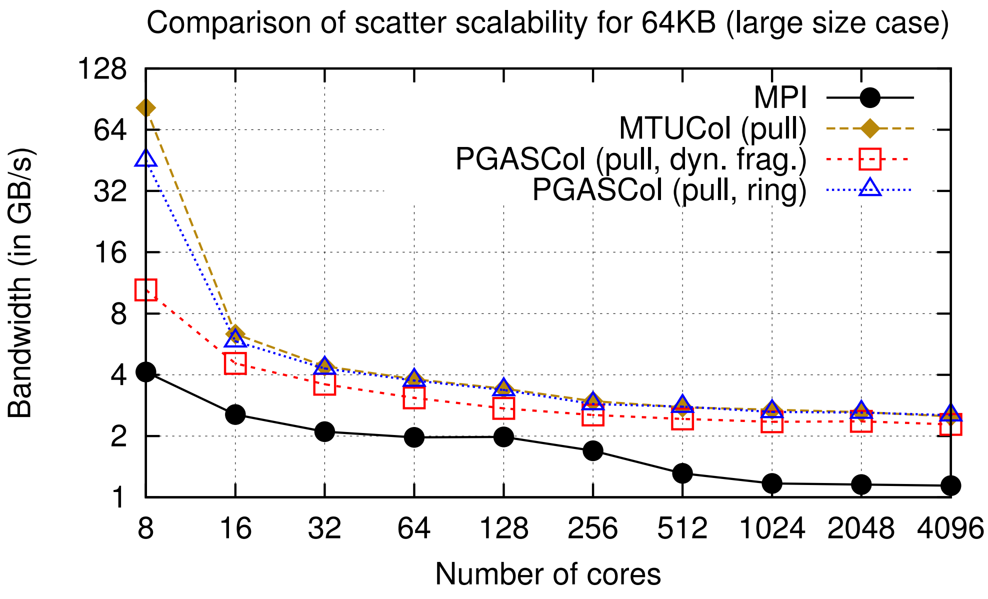
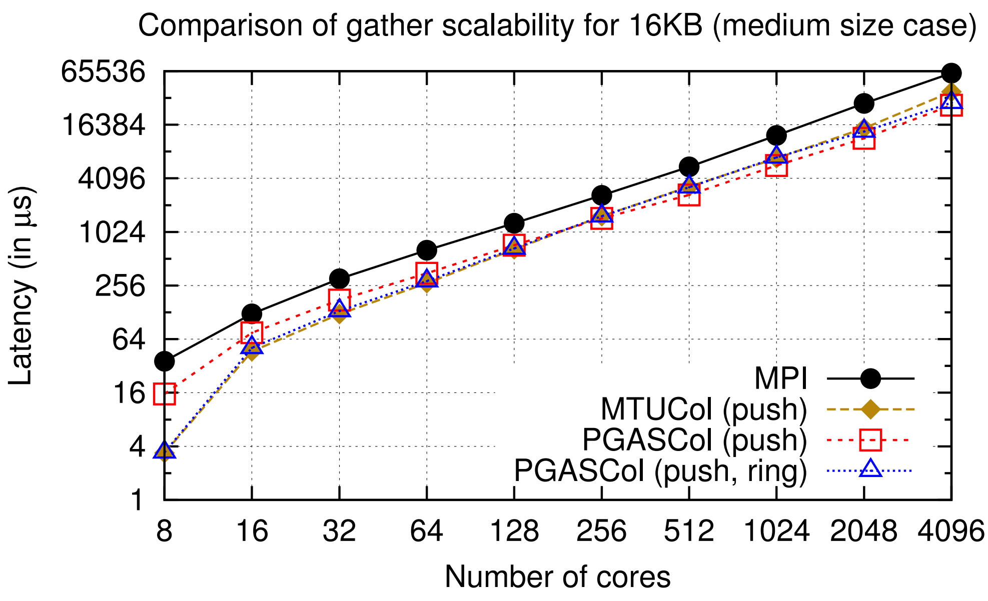


Figure 5.28: Comparison of scatter scalability of NUMA-based algorithms against MPI on JuRoPA



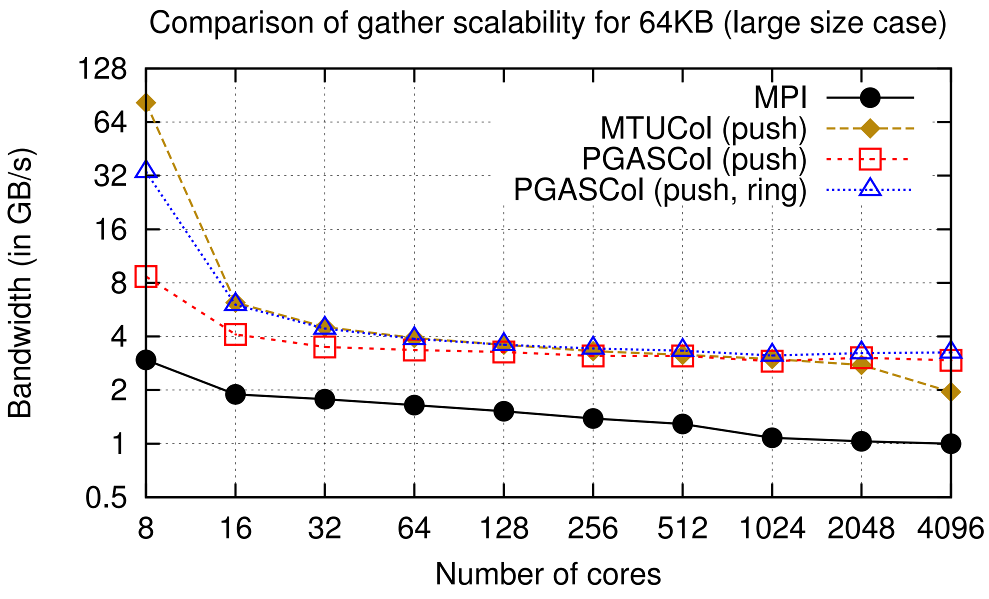


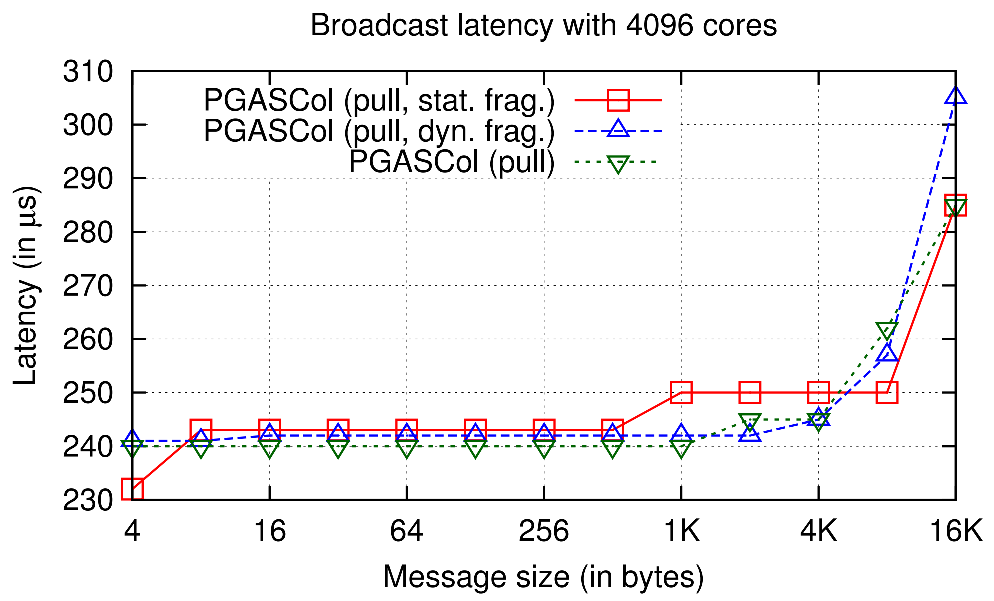
Figure 5.29: Comparison of gather scalability of NUMA-based algorithms against MPI on JuRoPA

aspect for achieving more performance in UPC operations is the efficiency achieved in shared memory thanks to the NUMA binding and the overlapping of communications possible with the push approach. In gather, the data flows upwards, causing the algorithm to be more sensitive to jitter and accumulating the penalty of ignoring optimizations of the memory subsystem. In scatter this is less important, since the data is transmitted to the root, and immediately pulled by other root processes from other nodes, which minimizes the penalty of not optimizing the memory subsystem, instead of adding additional overhead.

## 5.7. Impact on Performance of Different Optimizations at High Core-Counts

The basic algorithm has been optimized using different techniques. However, up to now, the contribution of each optimization to the overall collective performance has not been assessed. This section analyzes the influence of several optimizations for broadcast, due to its importance in the context of this work. The analysis has been focused on the impact of these techniques on scalability, using the minimum latency and the maximum bandwidth. Therefore JuRoPA has been selected for this analysis due to its higher number of available cores (it has been used up to 4096 cores).

Figure 5.30 shows the contribution of the message pipelining to the overall performance, for a setup of 4096 cores. In short message communication, with messages from 4 bytes to 8KB, the performance of the different variations of the algorithm shows the same performance results. In fact, they are using the same algorithm since the dynamic fragmentation algorithm processes messages larger than 8KB, and the static fragmentation algorithm starts processing messages larger than 32KB. It is from this point, messages larger than 32KB, that each algorithm presents a different performance. Thus, the pull algorithm without fragmentation increases performance slightly for 64KB and 128KB, achieving at this latter point its peak performance, degrading performance from that point on. The pull algorithm with dynamic fragmentation performs twice as good as the pull algorithm. However, its performance also degrades for messages larger than 256KB. The usage of the static fragmentation



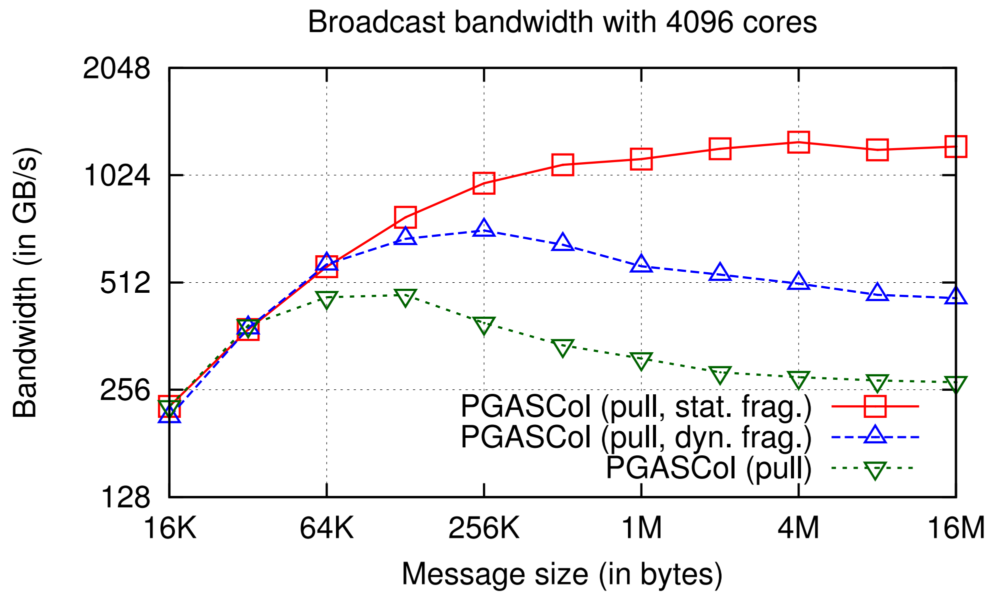
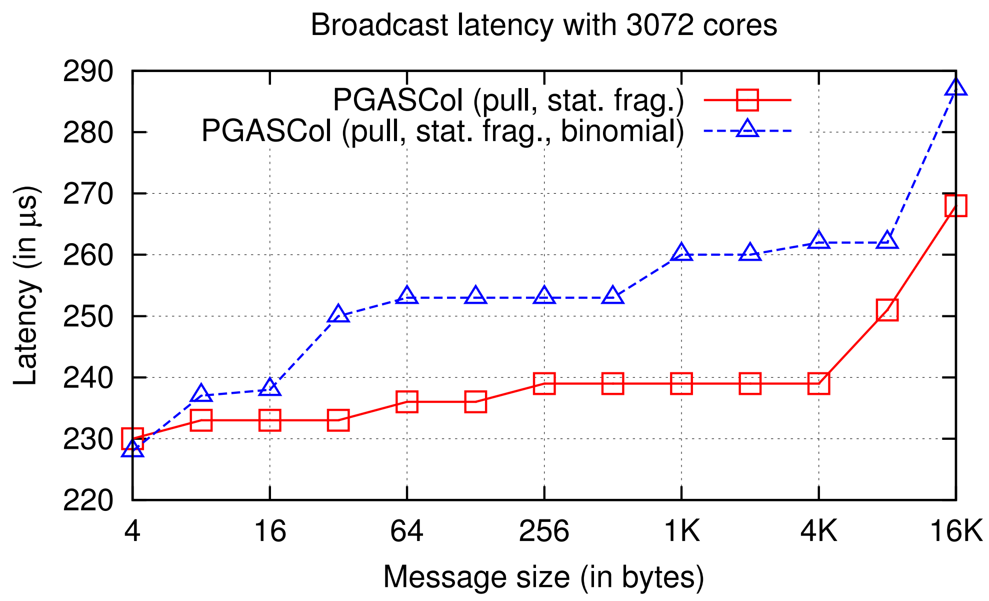


Figure 5.30: Impact of message pipelining in broadcast performance

pull algorithm achieves even higher performance, reaching its maximum at 4MB. At this point its performance is more than 4 times as good as the initially considered pull algorithm, showing the importance of message pipelining.

Figure 5.31 presents the impact on performance of the usage of multilevel trees. This experiment has been conducted with 3072 processes, with 512 nodes and 6 processes per node. A multilevel tree assigning 4096 processes, with 512 nodes and 8 processes per node, is equal to a standard binomial tree, due to the usage of a number of processes per node that is a power of 2. However, nowadays is increasingly common to find systems with a number of cores per node that is not a power of 2. It is in these scenarios where the usage of multilevel trees become important and where they are different from binomial trees. In scenarios where the short messages latency dominates the overall performance, the importance of having a multilevel tree is noticeable for messages larger than 16 bytes. The difference between both approaches is small up to 1MB. At that point the benefits of using the most efficient multilevel tree become more apparent as the message size increases, and for 16MB the use of a multilevel tree performs 1.5 times better than using a binomial tree.

The benefits of NUMA affinity to control the mapping of processes to the underlying hardware are negligible in setups with a high number of nodes where the effects of network latency and bandwidth have much more impact on performance than the small benefit obtained from NUMA binding control. Nevertheless NUMA affinity has shown its importance in shared memory scenarios. Moreover, a few facts suggest that NUMA affinity control has room for improving collective operations performance over the coming years: (1) the latest processor models are directly connected to network interfaces, typically one per node. In this case the relevance of having the node root process in the processor with direct connection to the network increases. Moreover, (2) the increasing number of NUMA regions per socket is forcing the consideration of new algorithms that are able to minimize jitter. The NUMA aware algorithms have outperformed other approaches in single node setups, with fully populated nodes. Finally, (3) as interconnection networks become faster, supercomputers with a high number of nodes turn out to be more sensitive to jitter. These facts suggest that NUMA affinity can have a major impact in collective performance in future systems. Moreover, affinity should be carefully evaluated for every application, as show in [40]. Correct affinity can have a significant impact on



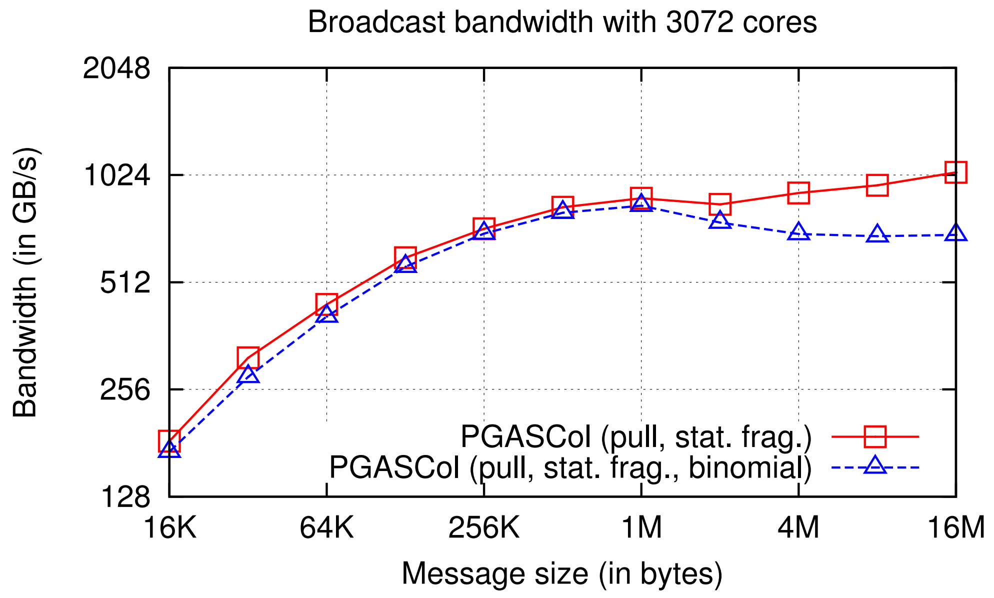


Figure 5.31: Impact of multilevel trees in broadcast performance

the performance of an application. The optimal affinity setup for any application will not interfere with the performance of PGASCol, as long as the trees are set up according to the process mapping.

## 5.8. Conclusions of Chapter 5

This Chapter has analysed the performance of the proposed PGAS algorithms for representative collective operations, particularly broadcast, reduce, scatter and gather, on 5 different systems (Finis Terrae, Superdome, SVG, JUDGE and JuRoPA), 4 different processor architectures (Intel Itanium 2, AMD Opteron Magny-Cours, Intel Xeon Westmere and Intel Xeon Nehalem) and 4 different interconnects (InfiniBand 4x DDR, Superdome Interconnect, Gigabit Ethernet, InfiniBand 4x QDR). The algorithms have been compared with the performance of an MPI implementation based on MPICH2.

The analysis of the implementation of the proposed algorithms has shown: (1) the implementation of these algorithms is able to equal and even outperform an evolved and more mature UPC library (BerkeleyCol); (2) PGASCol can outperform in some scenarios the state-of-the-art implementation of their equivalent functions in MPI, bringing another algorithm to the mix, allowing more possibilities for autotuning and choosing the most appropriate algorithm in each situation; (3) major contributor factors to performance are a tree mapped to the underlying hardware considering all levels, message pipelining, communications overlapping with adequate (pull vs. push) one-sided point-to-point transfers; (4) it is hard to determine which is the optimal tree shape for each level, as it depends on the architecture and message size; (5) tree-based collectives are often outperformed by ring algorithms with communication overlapping, in operations where data have to be scattered/gathered from a single point; (6) Finally, NUMA binding does not improve significantly the performance in nowadays clusters, as the main performance bottleneck is the network overhead. However, due to its highly scalable design, it is expected that the performance benefits of the developed library will be higher in future systems with tens of NUMA regions.

New massively parallel architectures, such as Intel Many Integrated Core, are gaining importance in HPC. Therefore, next Chapter presents the adaptation of the algorithms proposed in this Thesis to this architecture, and evaluates their suitability for it.

# Chapter 6

# Performance Evaluation of PGAS Collectives on Manycore Systems

In this Chapter the proposed algorithms are tested in one of the largest manycore systems currently deployed (Stampede, ranked in position number 7 in the top 500 list of November 2013 [110]). Their performance is compared with the performance of MPI collectives in two different optimized implementations, and an additional comparison is made between manycore processors, in this case Xeon Phi, versus standard multicore processors. This evaluation focuses on Broadcast, Scatter and Gather. Reduce in UPC, as seen in Chapter 5 is largely limited by latency. Therefore it has not been considered, as latency is typically higher in Xeon Phi than in standard processors, and due to that the impact of the algorithm optimizations will be hidden by this extra latency. The use of manycore processors as Xeon Phi has some implications in every algorithm. Therefore, the same collectives were evaluated on two different MPI implementations (Intel MPI and MVAPICH2). Section 6.1 lists the algorithms implemented in these two MPI runtimes. Section 6.2 explains the experimental setup. Sections 6.3 and 6.4 analyze the results of the UPC and MPI collectives, respectively. Section 6.5 compares the results using Xeon and Xeon Phi processors. Section 6.6 compares the UPC and MPI results. Section 6.7 assesses the impact of the contention caused by using POSIX threads (pthreads) instead of processes to implement UPC threads on Xeon Phi. Finally, Section 6.8 summarizes the analysis of the results.

## 6.1. Algorithms Implemented in Intel MPI and MVAPICH2

Intel MPI has a set of algorithms for its collective operations. Some of them are common to different operations. There is no documentation that details the implementation of the algorithms in Intel MPI. However, keeping in mind that Intel MPI is based on MPICH2, it makes sense to speculate that some of its algorithms are based on MPICH2 or its derived implementations (in particular MVAPICH2).

The first algorithm mentioned in the documentation of Intel MPI is the binomial algorithm. This algorithm is present in MPICH2 and MVAPICH2, and consists basically in a binomial tree of processes, where the data is propagated top-bottom for broadcast and scatter, and bottom-up for gather.

There is a variation of the binomial algorithm in Intel MPI, called topology aware binomial. There is no description of this algorithm in the documentation. However, MVAPICH2 has an algorithm implemented as a k-nomial tree that builds the tree taking into account the topology of the nodes and network participating in the job. Therefore, seems reasonable to assume that the algorithm implemented in MVAPICH2 is also the algorithm implemented in Intel MPI, with the difference of the tree radix, that is 4 by default in MVAPICH2.

The next algorithm mentioned in the documentation of Intel MPI is the ring algorithm. This algorithm is not present in scatter or gather operations. Whereas it is possible to implement a broadcast operation using purely a ring, i.e. passing the data to the next process in a ring fashion, this is highly inefficient. However, again, MPICH2 and MVAPICH2 have an algorithm for broadcast where the data is scattered across the processes, followed by an allgather function. This allgather function can be implemented in a ring fashion, taking $p - 1$ steps.

There is also a topology aware version of this algorithm in Intel MPI, implemented also in MVAPICH2 following the same principles as the topology aware binomial.

The next algorithm in Intel MPI is the recursive doubling algorithm. The description for this algorithm is basically the same as for the ring algorithm, as it is also present in MPICH2 and MVAPICH2, with the difference that the allgather phase is implemented using recursive doubling, which takes $log_2(p)$ steps.

As for the ring algorithm, the recursive doubling algorithm also has a topology aware version in Intel MPI and MVAPICH2.

The last algorithm implemented in Intel MPI is the so called Shumilin's algorithm. However, no publicly available documentation exist about the details of this algorithm, and there are no other MPI implementations with algorithms that suggest any specific detail about it.

It should be noted that, even though an explanation of the algorithms in Intel MPI is not public and admittedly the above paragraphs are an speculation about their behavior, there is not reason to believe that the algorithms do not work as described, due to: (1) self descriptive name in most of the algorithms, (2) similarity to names of algorithms described in the literature, and (3) being largely based in open source implementations that include such algorithms.

## 6.2. Experimental Configuration

This performance evaluation has been carried out on the Stampede supercomputer [110] at TACC (Texas Advanced Computing Center), using up to 15,360 Xeon Phi cores (256 nodes). Stampede is the 7th more powerful supercomputer in the world as of November 2013. Each node has 2 Xeon E5-2680 processors with 8 cores each, clocked at 2.7 GHz, and 32GB of memory. It also has one Xeon Phi SE10P per node, with 61 cores clocked at 1.1 GHz and 8GB of memory. The Manycore Platform Software Stack (MPSS) version is 2.1.6720-21. Generally it is recommended to rely on hybrid parallelization with few communicating threads or processes to take full advantage of the Xeon Phi computing power in an application. In this particular case 60 cores have been used per Xeon Phi in order to maximise its computational power, taking the Xeon Phi to its limits. The interconnection network is InfiniBand 4X FDR (54.54 Gbps of theoretical effective bandwidth). The node architecture is sketched in Figure 6.1. In Xeon Phi the operating system running on the accelerator is bound to core 61. The processes have been distributed in a block fashion, with 60 processes per node, avoiding the core where the operating system runs. The backend compiler used is Intel icc 13.1.1. The MPI compilers and runtimes are Intel MPI 4.1.1 and MVAPICH2-MIC (based on MVAPICH2 1.9). The Intel MPI experiments used the CCL-Proxy optimization, that routes all the Xeon Phi communication through a proxy service running in the host processor. The experiments performed using MVAPICH2-MIC used the proxy optimization for up to 1920 processes. It was not possible to succesfully run the experiments with that optimization and more than 1920 processes. Moreover, MVAPICH2-MIC did not run reliably on hybrid mode, failing on most setups, specially using messages larger than 128 bytes. Therefore, its results are not reported. It should be noted that MVAPICH2-MIC is not yet officially supported on Stampede. MVAPICH2 results on Xeon have not been obtained, since its algorithms and runtime optimizations are similar to those available on Intel MPI. The UPC compiler and runtime is Berkeley UPC 2.16.2. As the Xeon Phi is not officially supported on this release of Berkeley UPC, a few modifications had to be done. `lfence`, `sfence` and `mfence` memory fences were removed, as the Xeon Phi memory model does not rely on those memory fences for ordering. Also, support for 128 bit atomics was disabled on GASNet, as the `cmpxchg16b` instruction is not available on Xeon Phi. With those tweaks, Berkeley UPC runs experimentally on Xeon Phi. Berkeley UPC offers the user the possibility of running UPC threads as real operating system processes, or as POSIX threads. In Stampede, using the Xeon Phi coprocessors, the use of POSIX threads is the only way to run tests up to 15360 cores, as the use of the 2 or more operating system processes crashed when running on 256 nodes. Therefore, the execution of the tests used a single process per Xeon Phi, with 60 POSIX threads (pthreads in Berkeley UPC runtime). This limitation, using a single process per node, impacts performance negatively, as demonstrated in Section 6.7. Thus, using 1920 cores, with 2 processes per node and 30 pthreads, shows almost twice the performance than using a single process and 60 pthreads, using the broadcast pull based algorithm with dynamic fragmentation and flat trees at the node level, with message size of 1MB. This is due to the fact that the underlying GASNet layer has to be thread safe, and therefore the overhead for locks increases with the number of pthreads used. It is important to note that all threads participate actively on the collectives, as the distinction between process and thread is abstracted by the Berkeley UPC runtime, offering to the upper layers just the notion of UPC thread. A new version of Berkeley UPC (2.18.0), with official support for Xeon Phi, has been released after conducting these experiments. In order to ensure the validity of the results of the experiments some of them were repeated with the new runtime, choosing the most sensible setup, i.e.: 15360 cores. No significant differences have been observed, and it is still not possible to run more than one process per node –and less pthreads per process– when using 256 nodes.

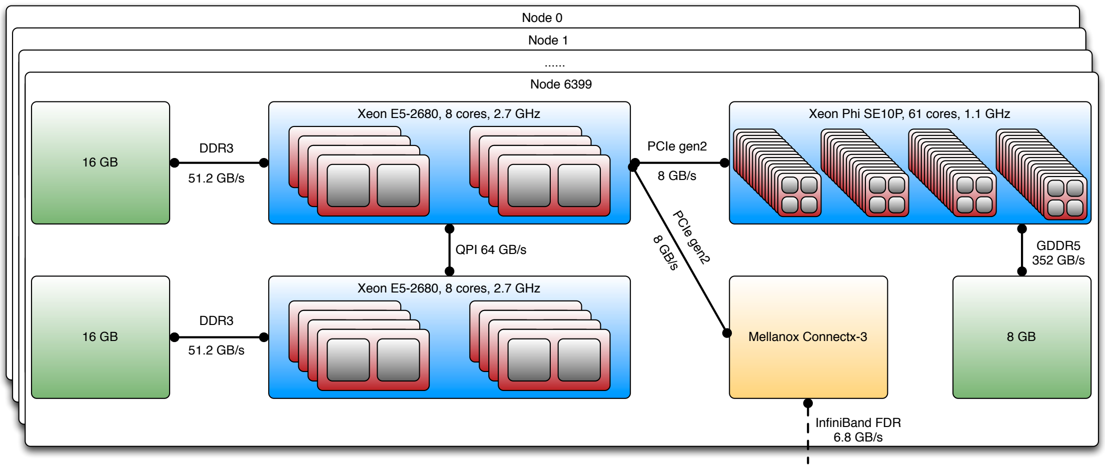


Figure 6.1: Stampede node architecture

Since the comparison between the PGAS collectives and MPI collectives is an important part of these subsections, the reported values for IMB are the maximum, i.e. the highest average time among processes, to guarantee a state where all the processes have finished the operation. The reported values for UOMS are the average, i.e. the average time per iteration needed to guarantee that all the processes have finished the operation.

All the MPI implementations have a default algorithm. Which one depends on the specific implementation, tuning and experiment, as the default algorithm can be different for different message sizes and number of processes involved. The default algorithm on Intel MPI will be evaluated to assess the suitability of the current thresholds for a Xeon Phi environment. The default algorithm on MVAPICH2-MIC will be evaluated to assess the performance of an implementation specifically optimized and tuned for Xeon Phi.

Sections 6.4 and 6.3 present the performance results of three representative collectives, broadcast, scatter and gather. Broadcast figures present the performance of a representative medium size message (16KB) on the top and the performance of a representative large size message on the bottom (1MB). Scatter and gather reported results have been obtained with shorter messages, 8 bytes and 1KB as representative sizes. With large number of cores the message size tends to be shorter, and it is limited by the memory requirements in the root process, which is the result of multiplying the message size by the number of processes. The $y$ axis represents latency in microseconds in the graphs on the top (medium size message case for broadcast and small size message case for scatter and gather), whereas the $y$ axis represents bandwidth in GB/s or MB/s in the graphs on the bottom (GB/s for large size message case for broadcast and MB/s for medium size message case for scatter and gather). As described in Section 5.1 of Chapter 5, variations in the same basic algorithm can lead to some dramatic performance differences. The graphs display only the most relevant algorithms for each combination of function and message size, giving more importance to the setups with high number of cores.

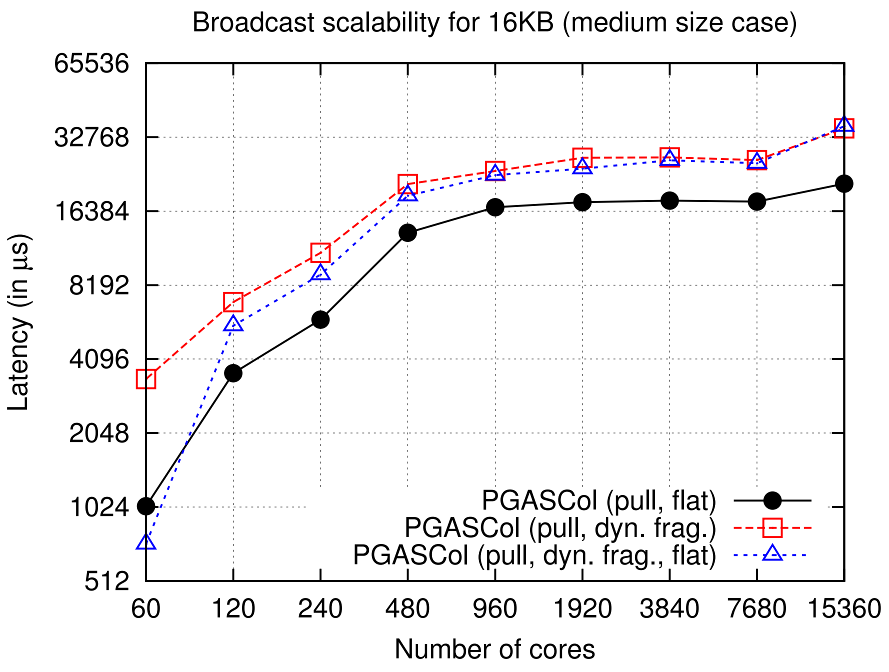
Additionally, Section 6.5 compares the performance of the PGAS collectives and MPI collectives on Xeon, Xeon Phi, and hybrid setups (using Xeon and Xeon Phi on the same experiment), comparing the results using fully populated nodes.

## 6.3. UPC Collective Performance Scalability on Xeon Phi

Figure 6.2 shows the performance of different broadcast algorithms implemented in UPC. In this case, the best algorithms are variations of the pull approach. In particular the pull algorithm with flat trees at the node level and without pipelining, the pull algorithm with dynamic fragmentation and binomial trees in the node level, and the pull algorithm with dynamic fragmentation and flat trees in the node level. In the medium size case stands out the behavior of the three selected algorithms, that reach a plateau at around 960 cores, showing a very good scalability, as adding more cores to the operation does not impact on the latency. This is due to the tree built to connect different nodes. Going from node 1 to 2 (60 cores to 120) has a big impact, as communications are going through the InfiniBand network, rather than exclusively through shared memory. However, the latency increases slowly, as adding an extra level to the tree is not meaningful. Stands out also the better performance of the pull approach with flat trees, which seems to be more adequate for medium messages than the other algorithms. In the large message case the performance of all the implemented algorithms is similar with 15380 cores. Nevertheless, the pull algorithm with dynamic fragmentation and flat trees is slightly better, as in large messages pipelining becomes more important.

Figure 6.3 presents the performance of the scatter operation. The best performing algorithms are all pull based, with binomial trees at both levels. The 3 selected algorithms show similar scaling in the small message case. In fact, in this scenario these algorithms are essentially the same for a number of cores smaller than 3840 (for the dynamic fragmentation algorithm, 15360 for the static fragmentation algorithm), since there is no message fragmentation for aggregated messages smaller or equal to 8KB (32KB in the static fragmentation case). However, they perform differently. This suggests that the small differences in the implementation that calculates offsets, number of chunks to complete a message and their size have an impact larger than expected, an effect that probably has been augmented by the slow cores present in Xeon Phi. Here the best performer is the algorithm with static fragmentation. However, in the large message case this algorithm is heavily penalized when scaling to thousands of cores because of the extra steps to synchronize the pipelining. All the algorithms have a drop in their aggregated bandwidth when going from 60 cores (1 node) to 120 cores (2 nodes), due to the impact of the use of the network, which is a major source of overhead in collective operations. In fact, the performance keeps dropping until 240 cores (4 nodes). From then on, for the pull algorithm and the pull algorithm with dynamic fragmentation the performance increases, as the messages sent through InfiniBand become larger and the usage of the network increases its efficiency.

Finally, Figure 6.4 shows the performance of the gather operation. Typically top-down collectives like broadcast or scatter benefit from a pull approach, whereas bottom-up operations like gather benefit from a push approach. However, one of the best algorithms in these experiments follows a pull approach. This confirms the trend seen in broadcast and scatter: the performance benefit of message pipelining seems to do not compensate the extra overhead involved. The slow cores and high start-up latency of small messages hinders the use of message pipelining. In the small message



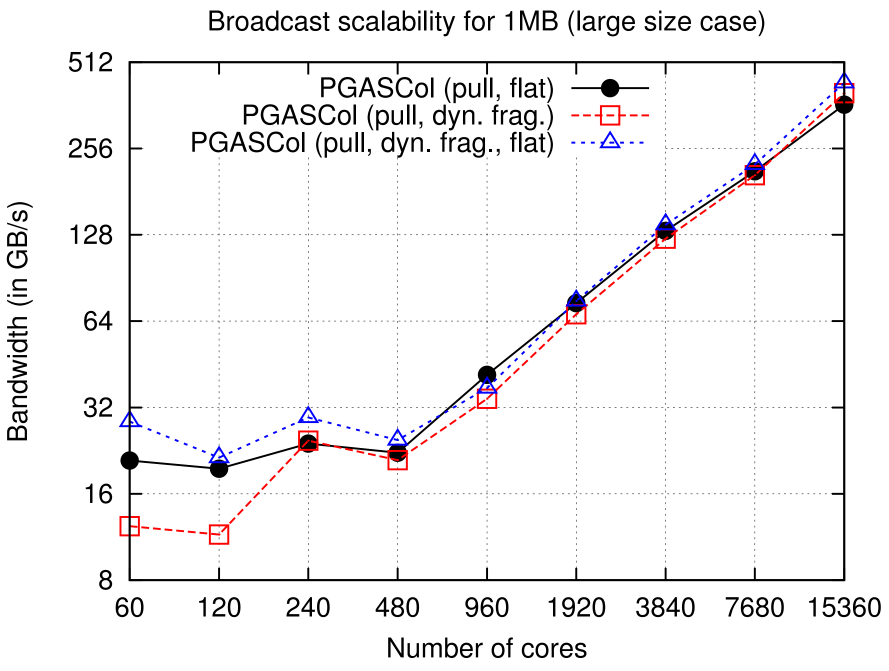
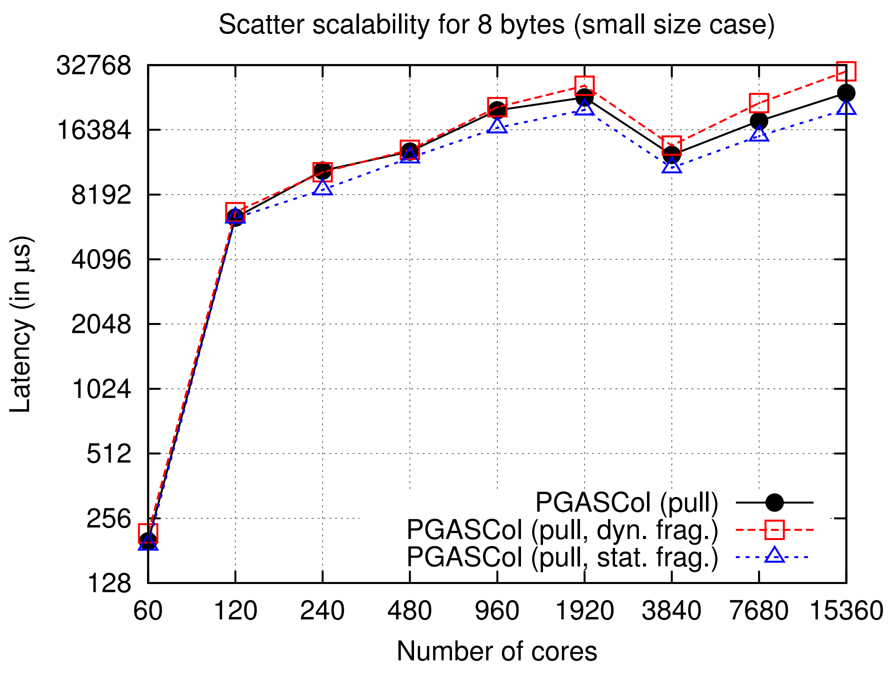


Figure 6.2: UPC broadcast performance and scalability on Xeon Phi



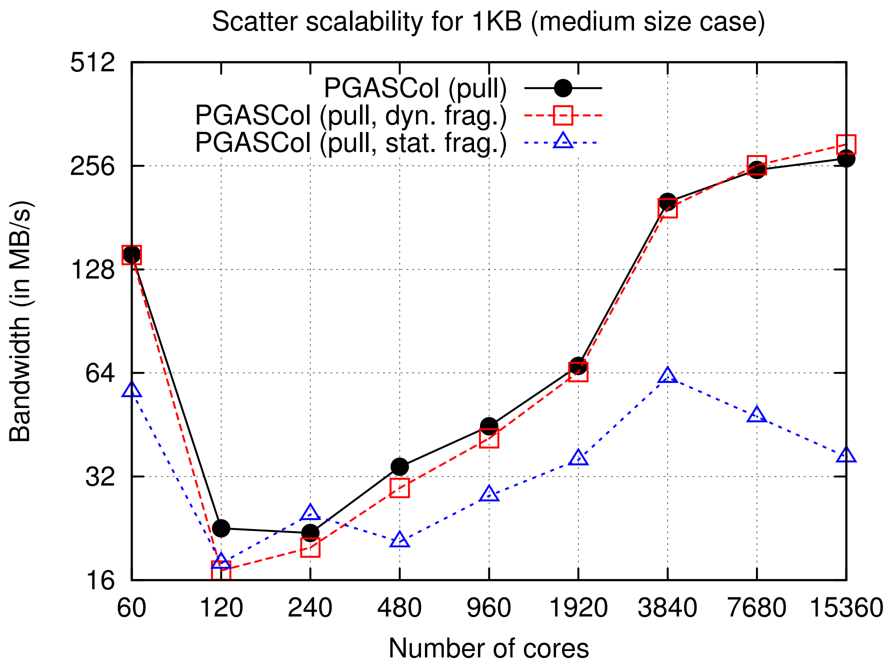
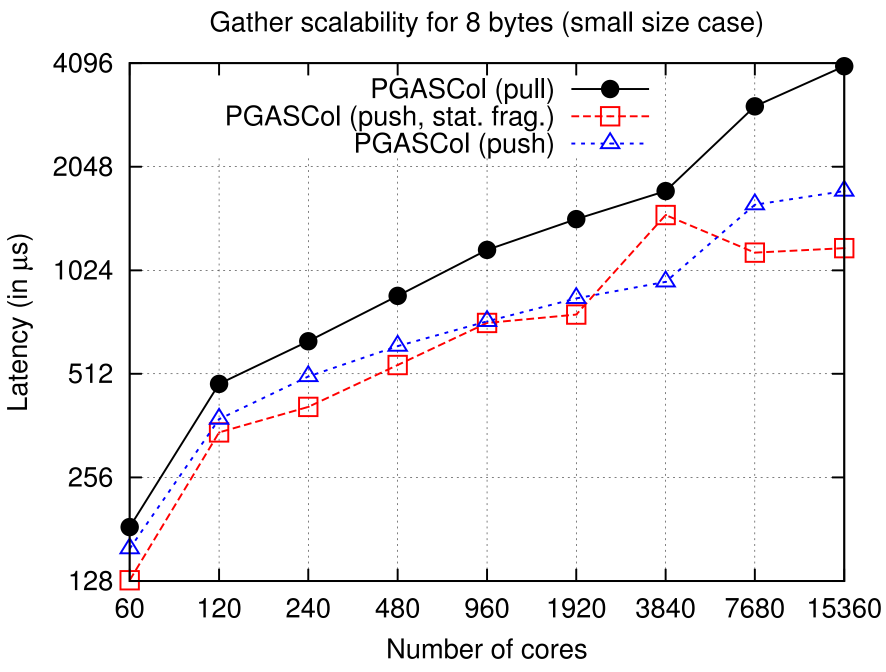


Figure 6.3: UPC scatter performance and scalability on Xeon Phi



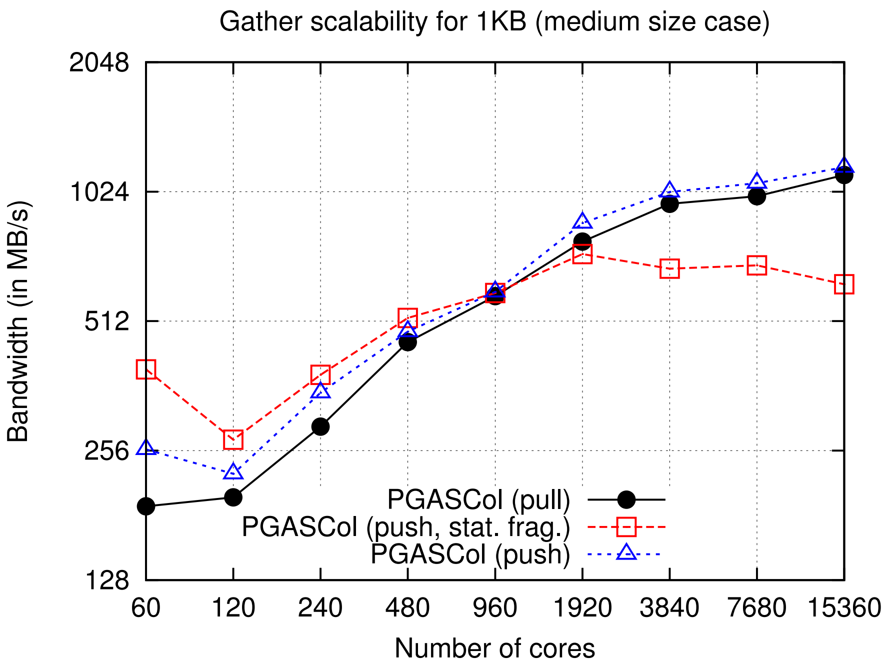


Figure 6.4: UPC gather performance and scalability on Xeon Phi

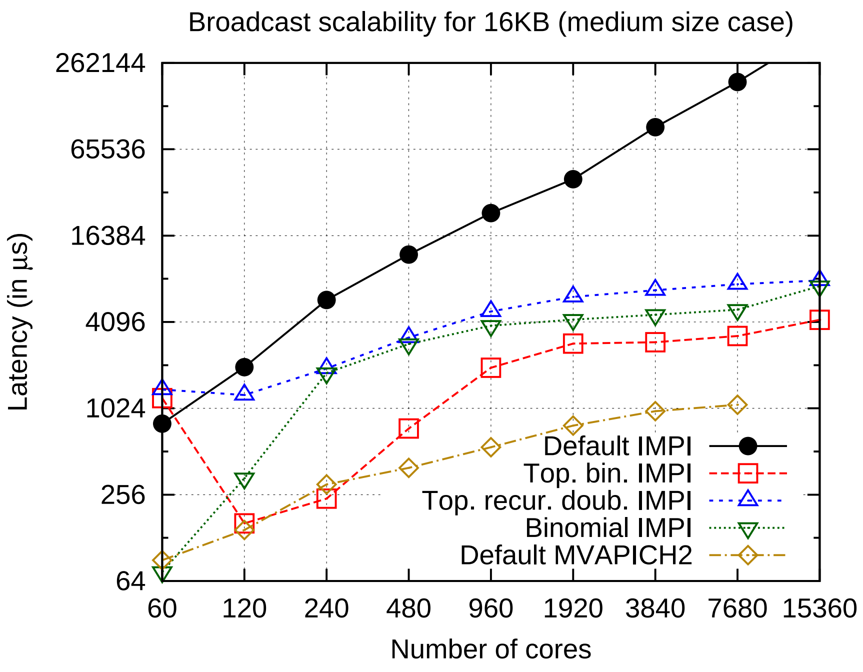
case the push approach scales worse than the other two algorithms, as expected. In the medium message case the push and pull approaches scale beyond 1GB/s, whereas the algorithm with static fragmentation stop scaling at 1920 cores (32 nodes), where the performance degradation due to fragmentation overhead reduces the aggregated bandwidth.

## 6.4. MPI Collective Performance Scalability on Xeon Phi

Figure 6.5 shows the performance of the broadcast operation for different MPI algorithms. In this case, the best algorithms are the binomial algorithm and the topology aware versions of the binomial algorithm and recursive doubling. Even though Shumilin's algorithm has very good minimum run time, the maximum (the value that determines at which point the collective is completed for all the processes) is higher than the remaining algorithms, and therefore it is not displayed in the plots. Stands out the poor performance of the default Intel MPI algorithm, that is showed here just for awareness of how important is to choose the right algorithm. In the medium size case the default MVAPICH2-MIC algorithm outperforms all the others, showing that the optimizations and tuning of the runtime are very effective on this scenario, even though the proxy support is not enabled for more than 1920 processes. Regarding Intel MPI the topology binomial algorithm is the one that exhibits the best performance, with a very flat increase in latency when using more than 8 nodes (480 cores). The binomial algorithm performs better than the topology aware recursive doubling, showing the suitability of tree-based algorithms for setups with small to medium sized messages and high core counts, where the recursive doubling algorithm cannot outperform the binomial algorithm, due to the small size of some of the transmitted fragments. In the large message size case, the observed results are different. MVAPICH2-MIC is not the best performer anymore, showing the importance of using the correct algorithm, regardless the specific runtime optimizations. Intel MPI outperforms MVAPICH2-MIC when using the correct algorithm. In this scenario the fragments transmitted by the recursive doubling algorithm are not that small anymore, and it slightly outperforms the binomial algorithms when using 15360 cores. It also stands out the change in the algorithm used by default by Intel MPI when using more than 16 nodes (960 cores), that leads to a significant performance boost, but far away from the best algorithms.

Figure 6.6 corresponds to the scatter operation. Shumilin's algorithm hangs for setups with more than 4 nodes, so its results are not reported. Therefore, for Intel MPI, besides the default algorithm, the relevant algorithms are the binomial and the topology aware binomial algorithms. The first result to notice is that the default MVAPICH2-MIC algorithm does not outperform the default Intel MPI algorithm for less than 1920 processes, on the small size case. The Intel MPI default algorithm behaves like the binomial for the whole range, which indicates that the default algorithm does not change depending on the number of cores or message size. In the small size case the binomial algorithm performs better than the topology aware algorithm for almost the whole range. The overhead of performing the algorithm in two phases outweighs the faster transfers, that are very fast due to the small size of the message. The binomial algorithm has a drop in performance when going from 960 cores to 1920, but keeps outperforming the topology binomial algorithm at the 15360 cores mark. For the medium size case all the algorithms drop in performance when going from 1 node to 2 (from 60 cores to 120 cores), as the network access becomes the bottleneck. Nevertheless, the default MVAPICH2-MIC algorithm keeps performing remarkably good for up to 1920 processes. With more processes the proxy optimization has not been used, and performance degrades sharply, proving its effectiveness in this case. The binomial algorithm on Intel MPI keeps improving its performance when increasing the number of cores, from 120, whereas the topology binomial algorithm keeps degrading its performance. In this scenario the topology binomial could be able to slightly outperform the binomial algorithm. Nevertheless, this is not the case. The measure of the experimental results has shown a very erratic behavior for this algorithm, with extremely differences between the minimum and the maximum for setups with more than 240 cores (being similar both values with 240 cores or less), and significant differences in performance between 64 and 128 bytes messages.

Figure 6.7 displays the gather operation. Gather is conceptually the inverse of scatter, and the analysis and observation done for the later is also valid here, where the algorithms are the same, and the observed behavior and performance are very similar. The exception to this statement is the performance of MVAPICH2-MIC in



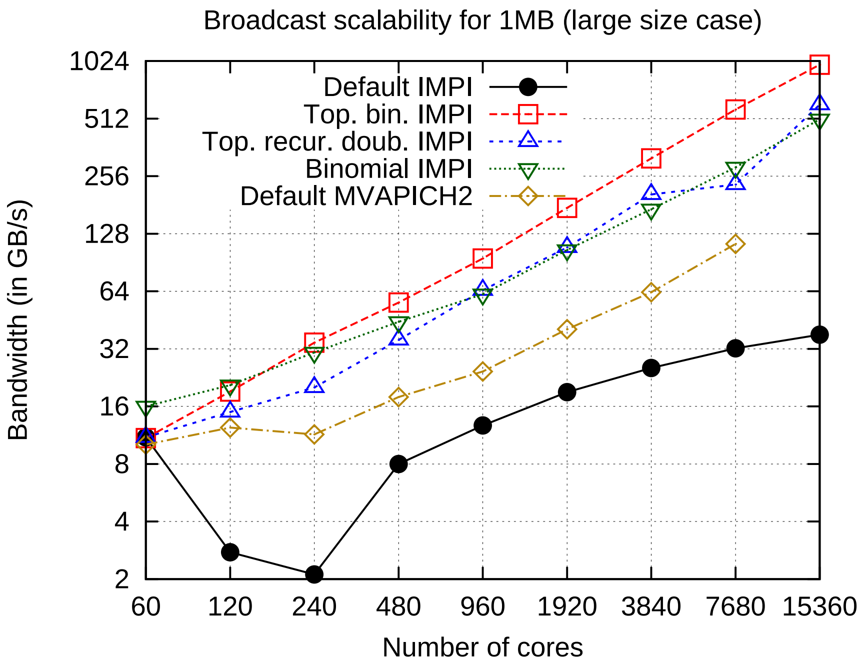
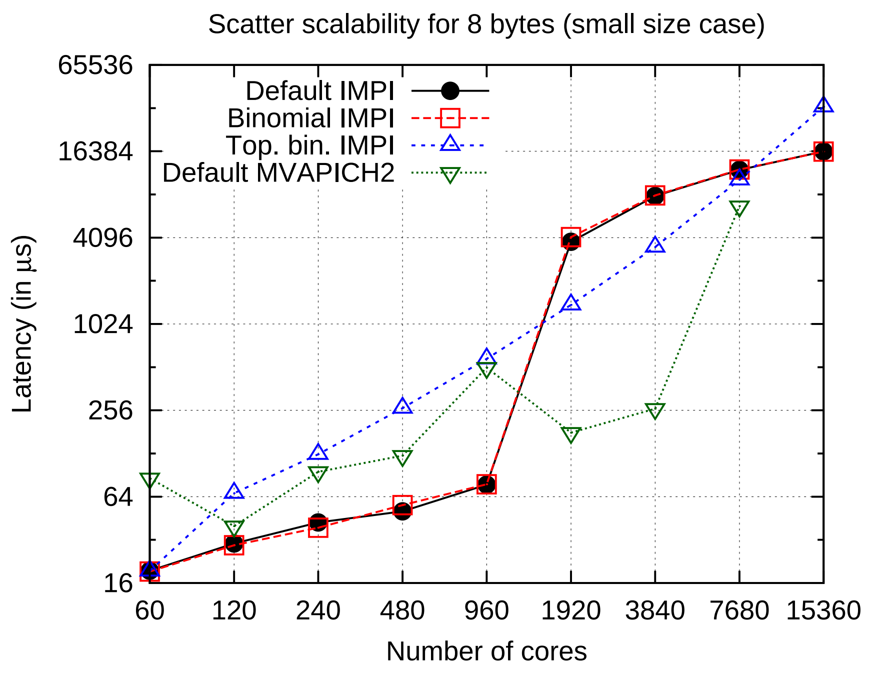


Figure 6.5: MPI broadcast performance and scalability on Xeon Phi



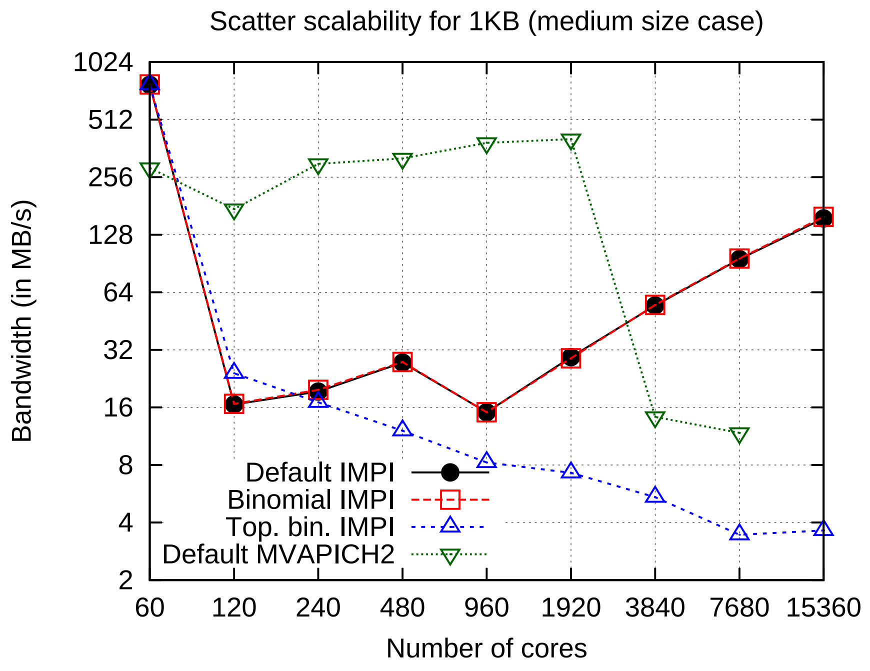
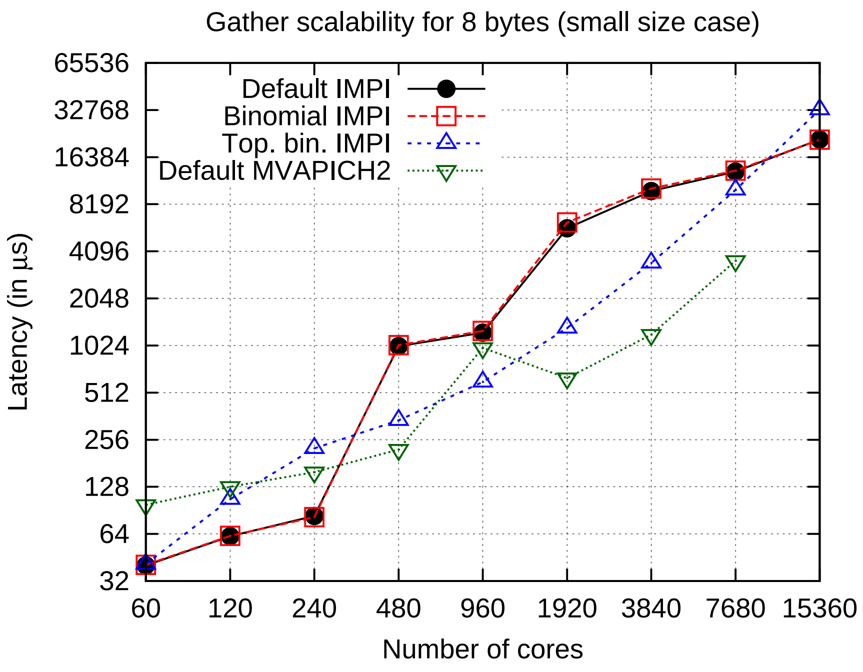


Figure 6.6: MPI scatter performance and scalability on Xeon Phi



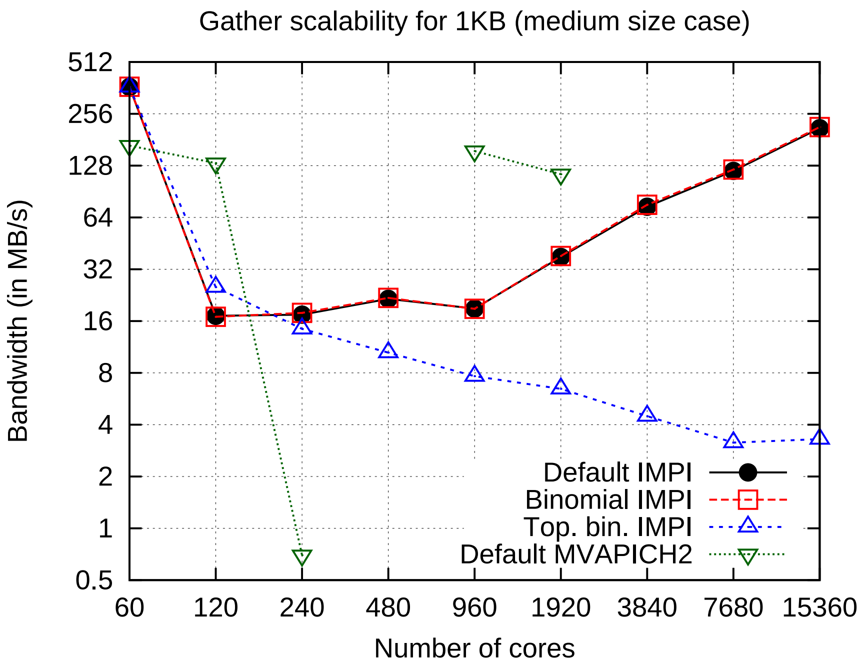


Figure 6.7: MPI gather performance and scalability on Xeon Phi

the medium size case, with a very bad performance for 240 processes and no results for 480 processes. However, for 960 and 1920 processes its performance is better than any of the algorithms implemented on Intel MPI.

## 6.5. Collective Operations Performance on Xeon versus Xeon Phi

This subsection compares the performance of collective operations in Xeon versus Xeon Phi. The purpose is to assess the impact of the processor used in communications performance. Figures 6.8, 6.9 and 6.10 show the results for UPC with the performance of the same algorithms on Xeon, using the same number of nodes, for broadcast, scatter and gather, respectively. Figures 6.11, 6.12 and 6.13 compare the performance of the most relevant Intel MPI algorithms on Xeon Phi. Results for MVAPICH2-MIC on Xeon Phi have been also included. The comparison has been made using the same number of nodes, which are fully populated, and therefore the number of cores will be different for Xeon and Xeon Phi. Hence, the Xeon Phi experiments use 60 cores per node, whereas the Xeon experiments use 16 cores per node. All the experiments have been carried out on Stampede, using from 1 to 256 nodes. For each figure, there are two graphs. The top graphs within the figures are focused on latency, whereas the bottom graphs are focused on bandwidth. The MPI figures additionally have results with hybrid setups, using both Xeon and Xeon Phi processors. These setups also use fully populated nodes, ranging from 76 to 19456 cores (1 to 256 nodes).

Figure 6.8 shows the performance of UPC broadcast. Regarding latency, the results confirm the observations made on previous experiments. The communication latency is much higher on Xeon Phi. Besides that, UPC latencies are always higher than MPI latencies. Again, Xeon Phi is more affected when using more than one node, with latencies significantly higher. These results confirm that the reason for the suboptimal performance of these algorithms on Xeon Phi is the high latency of the access to the network, as these algorithms rely extensively on short synchronization messages to coordinate the pipelining. However, the scalability is quite good, with an almost constant latency on Xeon Phi, from 8 to 256 nodes, whereas on Xeon it increases accordingly with the number of cores. The bandwidth obtained in one node is higher on Xeon Phi than on Xeon, thanks to the one-sided nature of the operations. The performance benefit is up to 2.4 times for the pull algorithm with flat trees and dynamic fragmentation. However, this advantage is lost when using more than one node as the impact of the low performance of Xeon Phi accessing the network is especially high. Thus, Xeon Phi performance results are less than half of the Xeon performance when using 256 nodes.

In Figure 6.9 the results for UPC scatter are presented. As for the other setups already discussed, the communication latency for Xeon is much lower than for Xeon Phi. However, as the latency on Xeon Phi remains constant from 16 nodes on, the latency on Xeon keeps increasing. Regarding bandwidth, the results for Xeon and Xeon Phi for the pull binomial algorithm are similar when using a single node, but the performance of Xeon Phi falls significantly when using multiple nodes because of the network access overhead. Here Xeon is the best performer, but Xeon Phi improves when using more nodes, as the bandwidth obtained increases at a better pace than on Xeon, except for the pull algorithm with binomial trees and static fragmentation. In this particular case there is a high number of messages, a scenario which is especially penalised by high start-up latencies and therefore its performance does not increase.

Finally, Figure 6.10 shows the results for UPC gather. It is remarkable the good performance of Xeon Phi with more than 16 nodes, where it performs similarly to Xeon, or even better. Gather is a bottom-top collective, where leaves of the tree operate independently. In other words, more parallelism is exposed at the beginning of the operation, and copies are done asynchronously towards the root, whereas for scatter less parallelism is exposed at the beginning, and transfers downwards have to wait for the data to arrive. This allows these gather algorithms to perform better than their scatter counterparts. The same effect is noticeable for bandwidth.

Figure 6.11 shows the results for the MPI broadcast. As for UPC, for MPI the latency is much higher on Xeon Phi than on Xeon. The higher number of cores in Xeon Phi contributes to the higher latency, as more data has to be distributed. However, it is remarkable that the Xeon with 256 nodes is faster than Xeon Phi in a single node (except for the binomial algorithm), even though the number of cores is much higher (4096 cores for Xeon vs. 60 cores for Xeon Phi). It is also highly

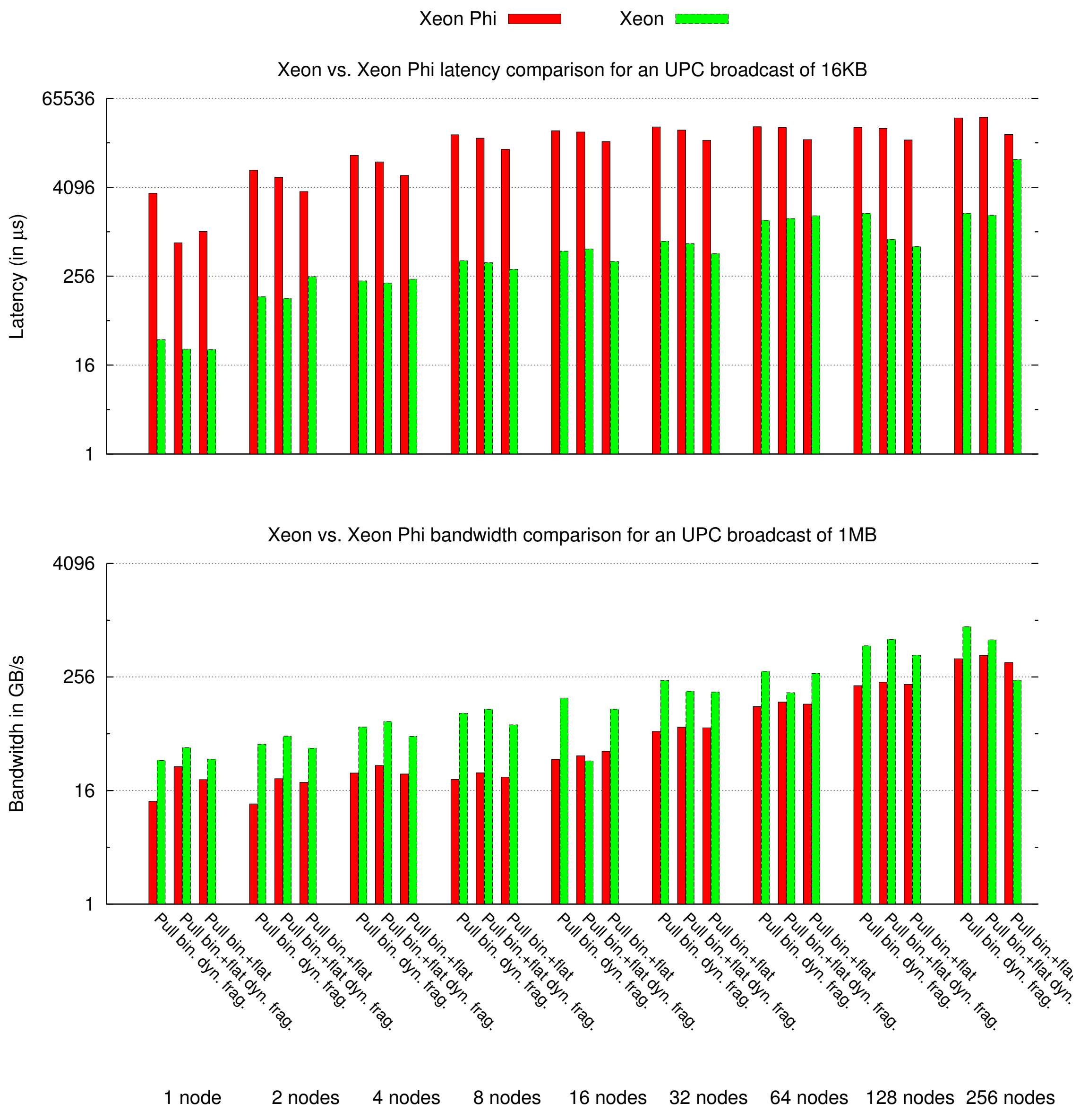


Figure 6.8: Performance comparison of UPC broadcast algorithms, for Xeon and Xeon Phi

remarkable the impact of the network, where Xeon Phi is heavily affected, especially on the default algorithm of Intel MPI, which performs poorly through all the tests. The hybrid Xeon + Xeon Phi setup is limited by the Xeon Phi coprocessors, and therefore its latency is almost equal to the Xeon Phi setup for all the cases. Regarding bandwidth the trend showed by the default algorithm on Intel MPI on

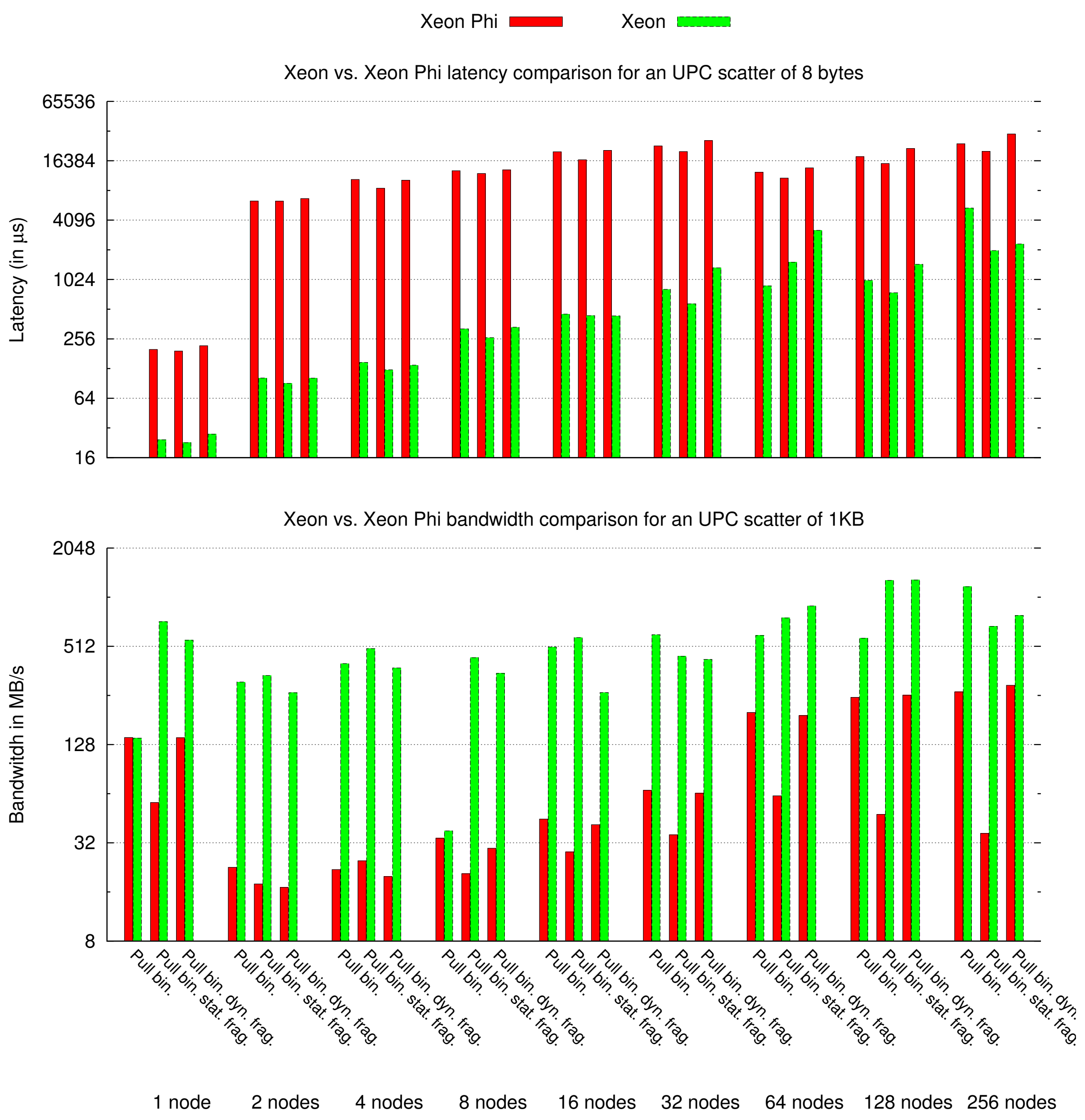


Figure 6.9: Performance comparison of UPC scatter algorithms, for Xeon and Xeon Phi

Xeon is to scale its performance with the availability of more resources, whereas on Xeon Phi it slows down up to 32 nodes, point where the default algorithm improves, showing that the default thresholds are not set properly on Xeon Phi. The binomial algorithm with topology awareness increases its performance in both Xeon and Xeon Phi, as well as the recursive doubling algorithm with topology awareness.

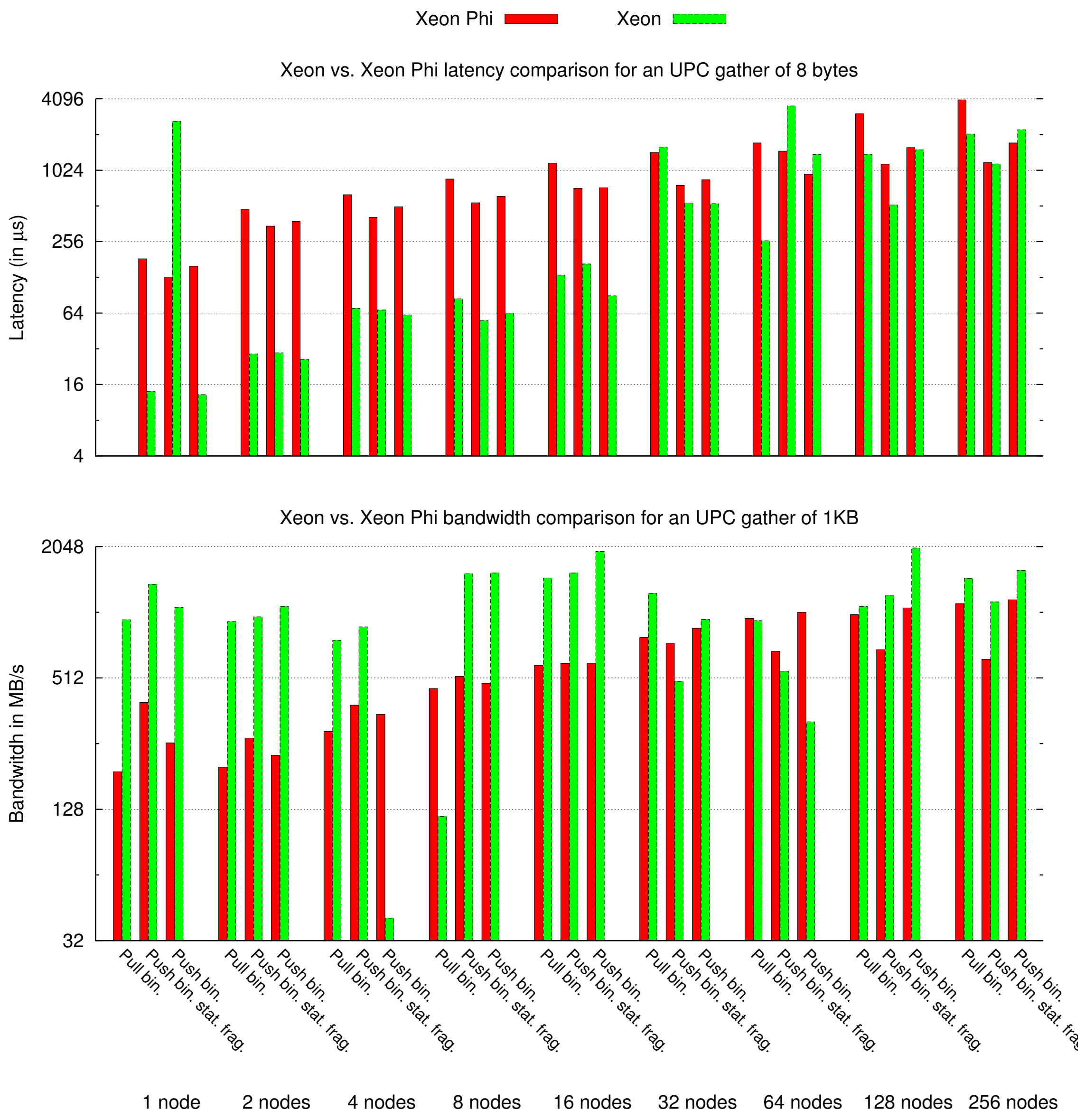


Figure 6.10: Performance comparison of UPC gather algorithms, for Xeon and Xeon Phi

The performance of the binomial algorithm is very similar to the performance of the topology aware binomial. As a general conclusion, the Xeon results are significantly better than the Xeon Phi performance numbers. The performance of hybrid setups sits between the performance of Xeon and Xeon Phi.

Figure 6.12 presents the results for MPI scatter. Latency wise the trend is

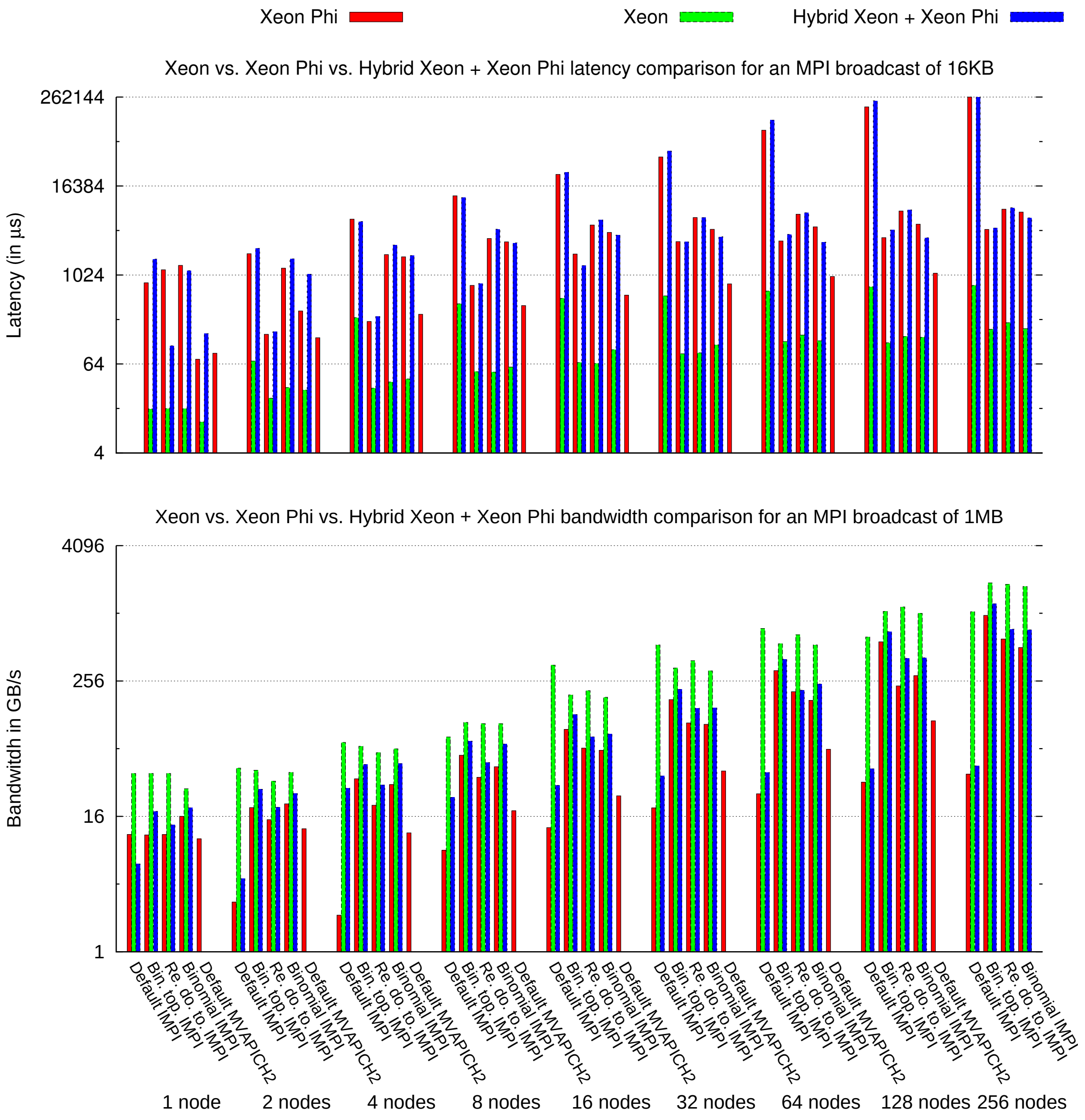


Figure 6.11: Performance comparison of MPI broadcast algorithms, for Xeon, Xeon Phi and hybrid Xeon + Xeon Phi

the same as for broadcast. The Xeon experiments have lower latency than the experiments with Xeon Phi for every setup, especially in the binomial algorithm, since the overhead of calculating the optimal tree has a big impact in setups with small messages. Differently from the broadcast case, in scatter the hybrid setups have better latency than pure Xeon Phi setups. Nevertheless, it is still significantly

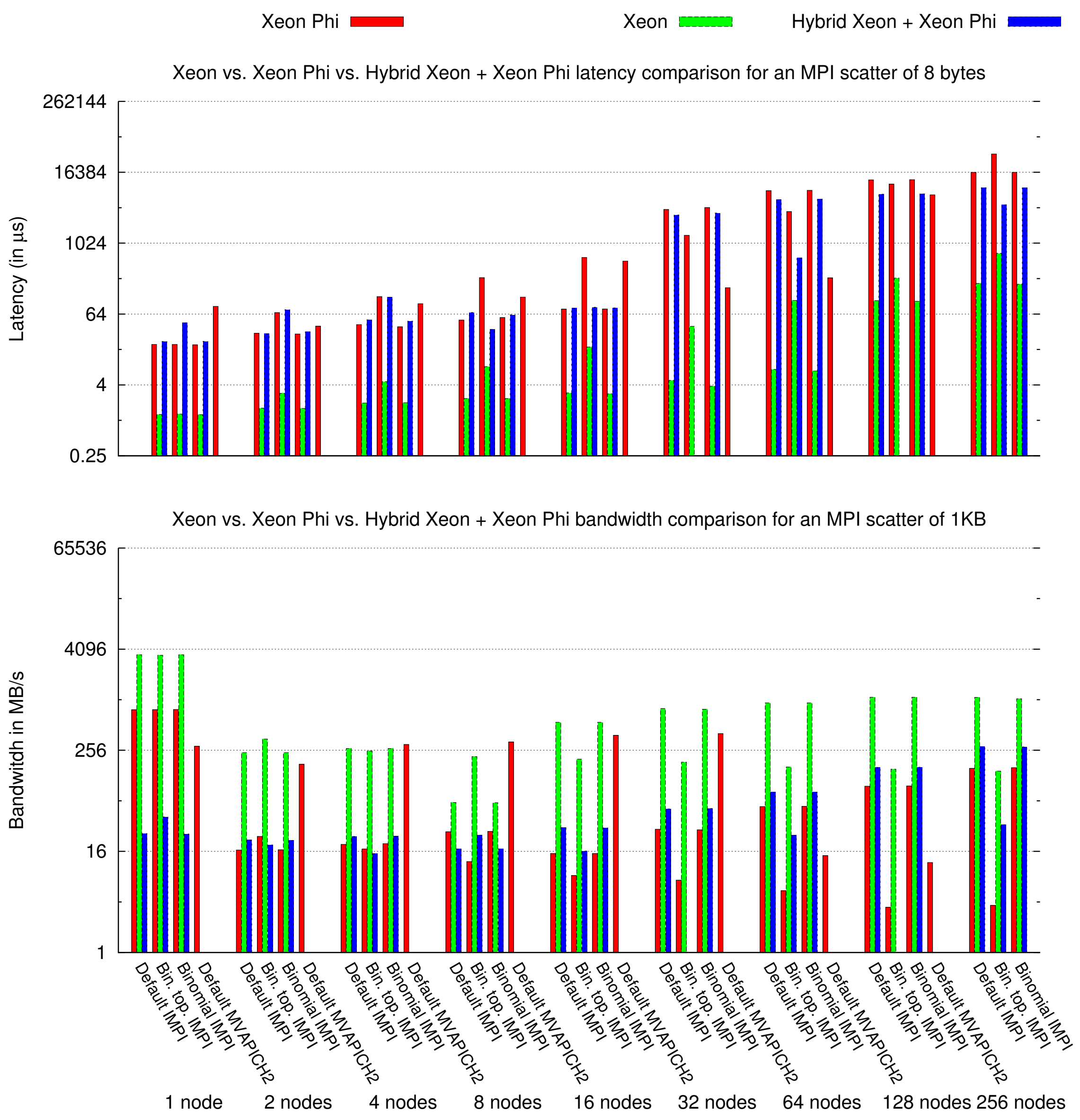


Figure 6.12: Performance comparison of MPI scatter algorithms, for Xeon, Xeon Phi and hybrid Xeon + Xeon Phi

higher compared to the results on Xeon, that performs more than 60 times better than Xeon Phi, for the default algorithm on Intel MPI using 256 nodes. On the bandwidth scenario Xeon is the best performer, with the hybrid setups between Xeon and Xeon Phi. Interestingly the topology binomial algorithm performs equal or better on Xeon than the binomial algorithm, for up to 8 nodes, whereas it is

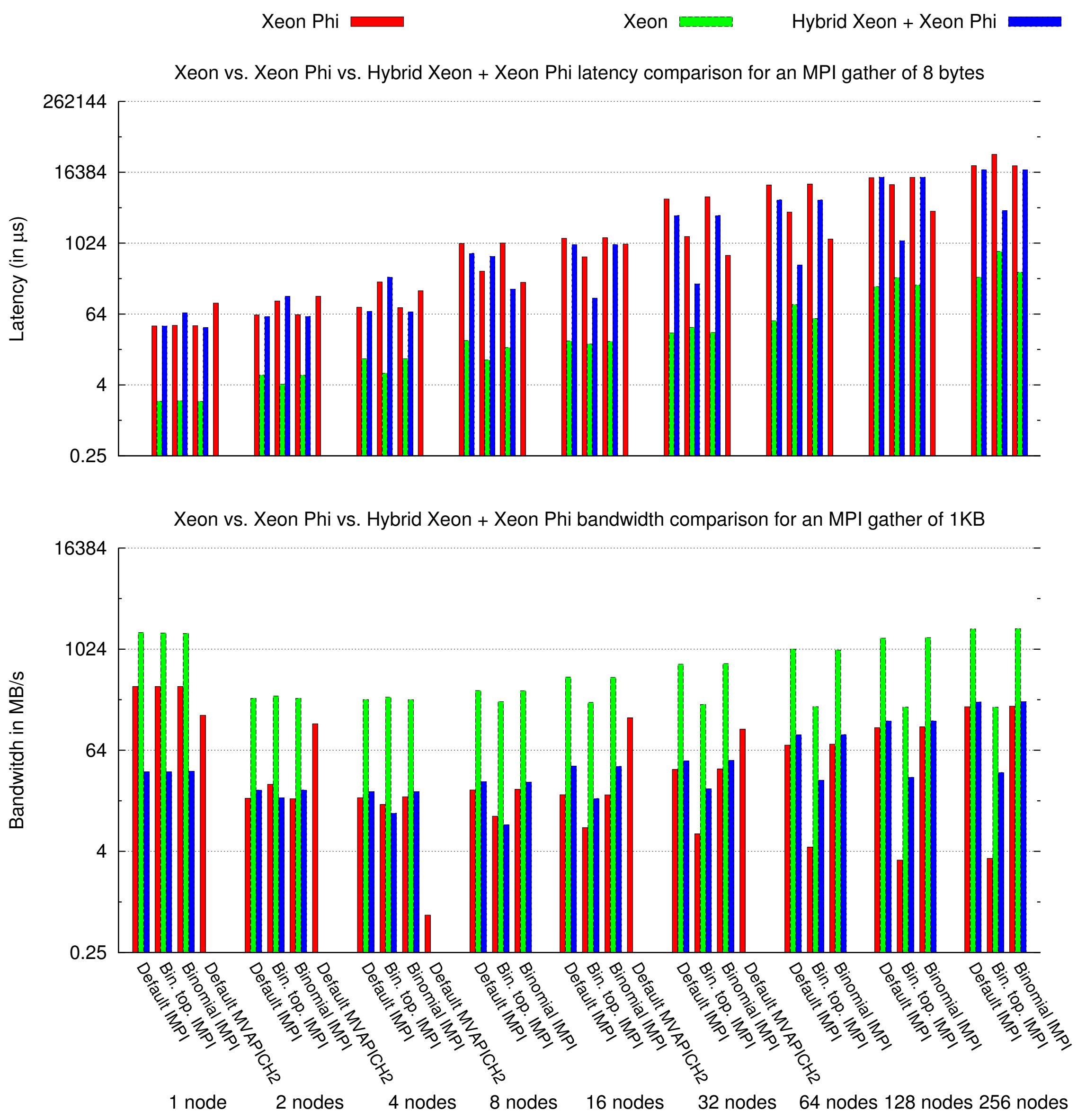


Figure 6.13: Performance comparison of MPI gather algorithms, for Xeon, Xeon Phi and hybrid Xeon + Xeon Phi

easily outperformed by the binomial algorithm in setups with more than 16 nodes. On Xeon the bandwidth reaches its peak already with 64 nodes, whereas on Xeon Phi the bandwidth keeps increasing slowly through all the setups.

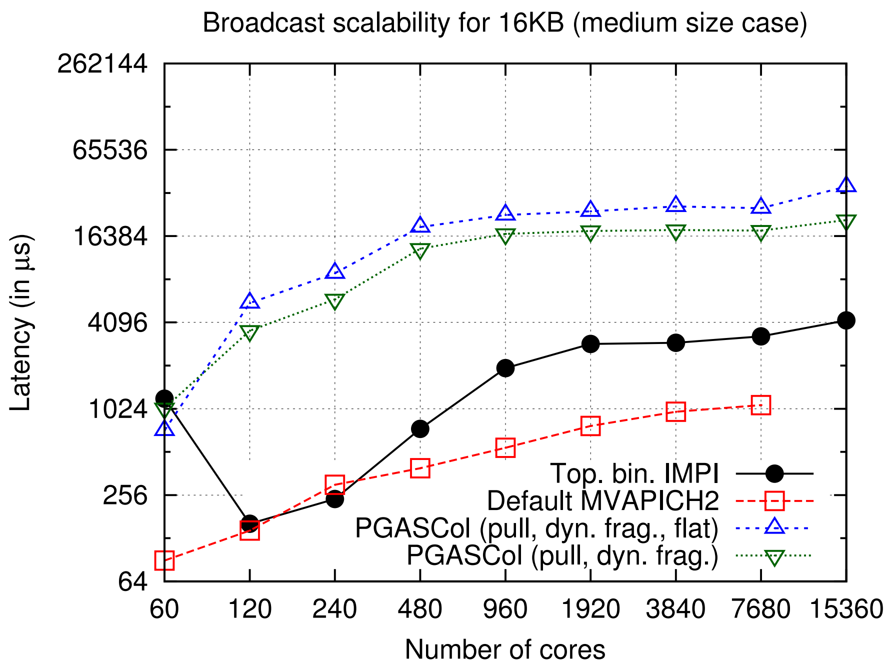
Figure 6.13 shows the results for MPI gather. Like the Xeon Phi analysis done in Subsection 6.4, the analysis for gather is quite similar to the analysis for scatter, with Xeon latencies much lower than on Xeon Phi. The high start-up latencies of Xeon Phi also limits the performance of the hybrid setups. Bandwidth is also much better on Xeon, with the performance of the hybrid setups slightly better than the performance of the Xeon Phi setups, except for the topology binomial algorithm, where the hybrid setups perform up to 8 times better than Xeon Phi.

## 6.6. UPC versus MPI Collective Operations Performance on Xeon Phi

This Section correlates the performance of the PGAS collectives with the performance of the MPI collectives on Xeon Phi. In the previous Section results were obtained also in Xeon processors. The comparison between UPC and MPI has not been done in Xeon due to the similarities of Stampede with JuRoPA, were this comparison was already made, even though with different MPI implementations. Besides this, and more importantly, running Berkeley UPC with 60 POSIX threads in a node –as done in Stampede– clearly harms the performance of the collectives, as demonstrated in Section 6.7. Therefore, making a fair comparison is not possible. Such comparison has been made on Xeon Phi to be able to have an initial estimation. However, in Xeon is not necessary due to the experiments carried out in Subsection 5.6.

Figure 6.14 shows the performance comparison of the broadcast collective. In this case, both MPI implementations outperform the PGAS collectives in the latency bound scenario. In this particular case, the proxy optimization of MVAPICH2-MIC does not have a clear benefit, as for more than 1920 cores it is not enabled, and yet its performance keeps the same trend. For large messages the PGAS collectives are remarkably better than MVAPICH2-MIC, even though both have a similar scaling characteristics. The topology binomial algorithm of Intel MPI scales like the other evaluated algorithms, but with a better performance.

Scatter is analyzed in Figure 6.15. In the small size case, the latencies of the PGAS algorithms are higher than the MPI algorithms. However, when using a large number of cores, latencies get much closer. For stands out the good performance of the pull with dynamic fragmentation and pull implementations when compared to



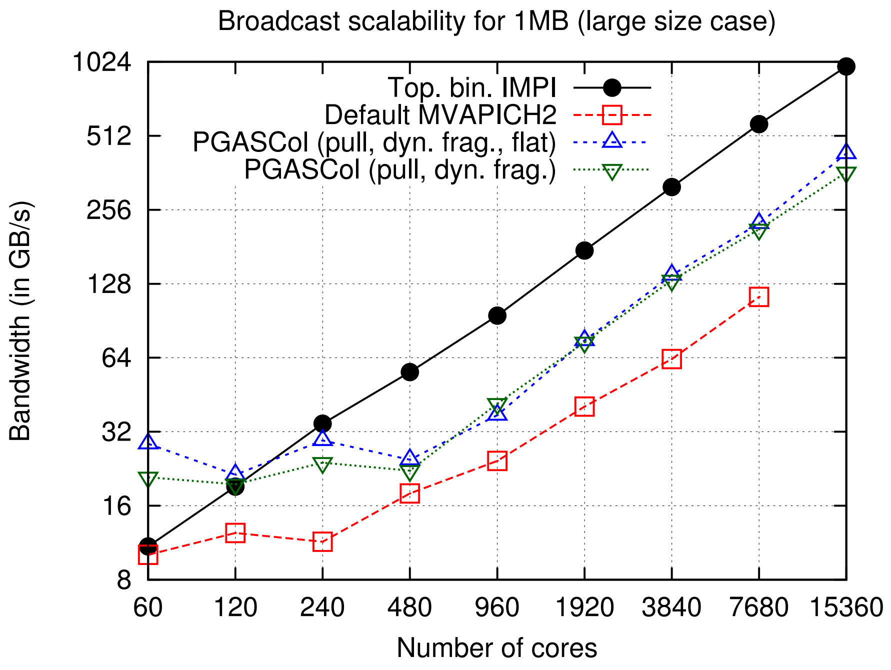
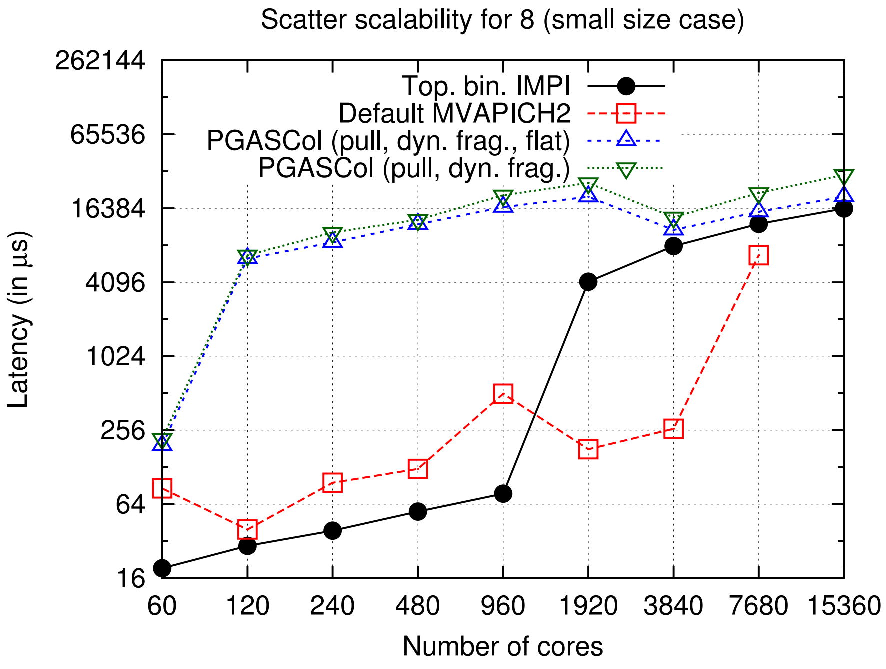


Figure 6.14: Performance comparison of MPI and UPC broadcast algorithms, for Xeon Phi



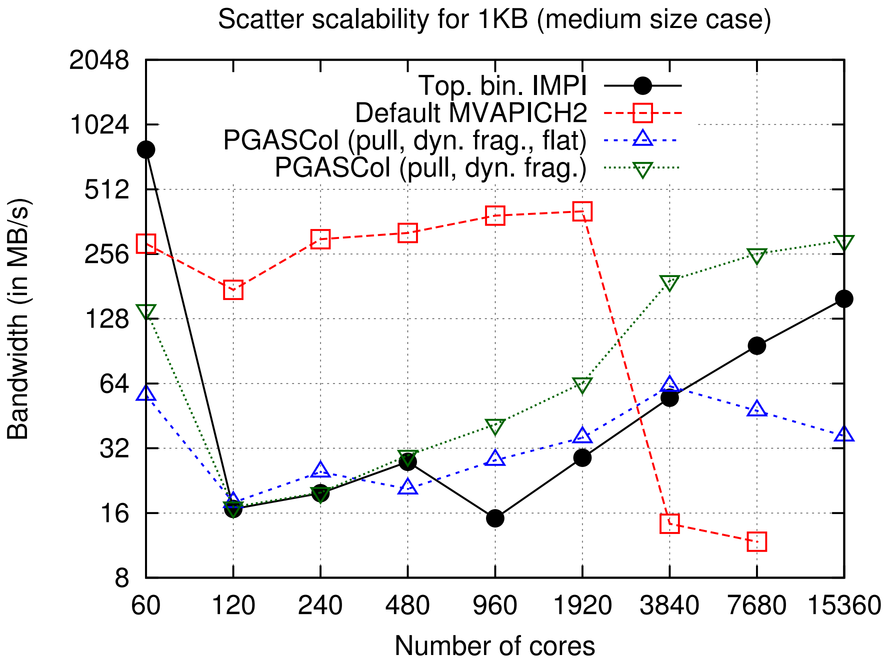
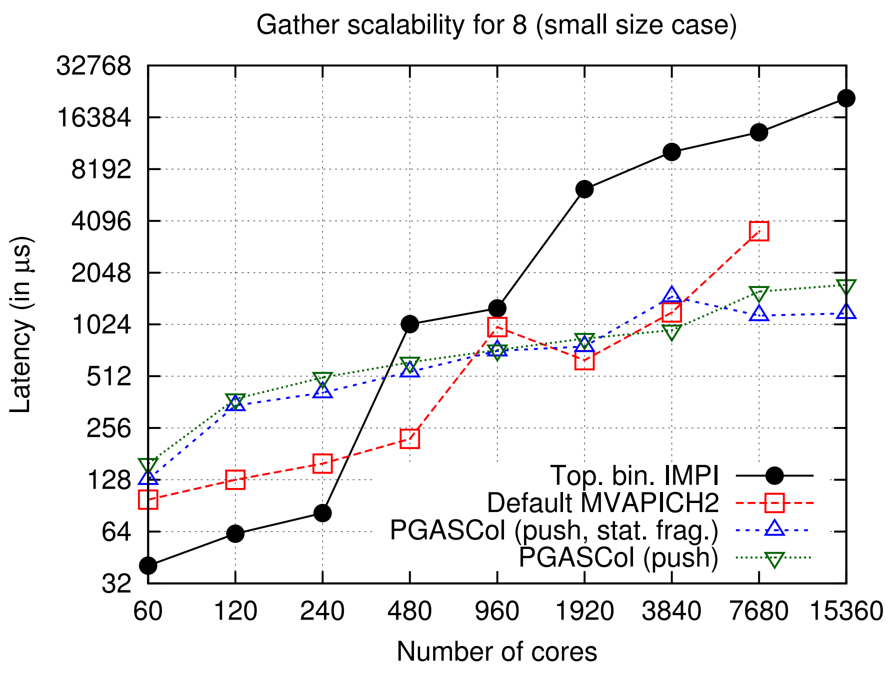


Figure 6.15: Performance comparison of MPI and UPC scatter algorithms, for Xeon Phi



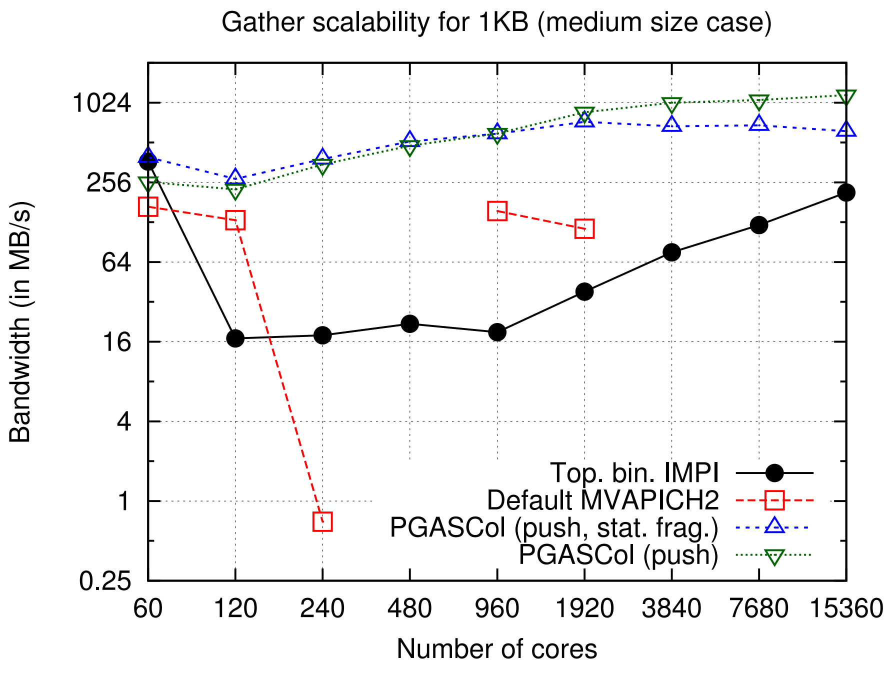


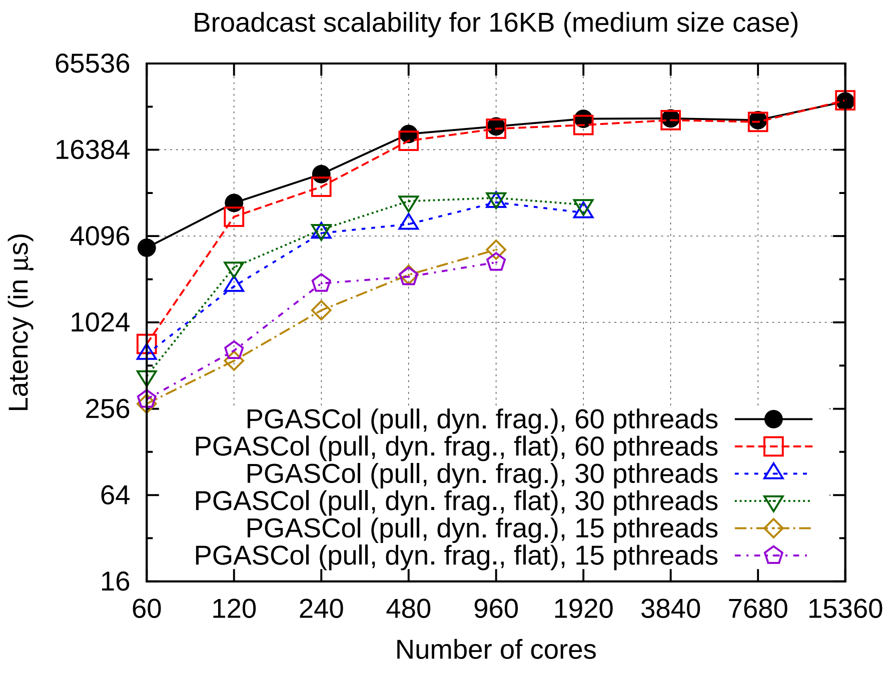
Figure 6.16: Performance comparison of MPI and UPC gather algorithms, for Xeon Phi

Intel MPI, as their bandwidth is up to 75% better with 15360 cores. No comparison can be done with MVAPICH2-MIC using 15360 cores, since MVAPICH2-MIC did not successfully run with that number of cores. However, the performance of MVAPICH2-MIC is better than any other algorithm when the proxy optimization is enabled ($\leq$ 1920 cores).

Figure 6.16 presents the results for gather. As with scatter, stands out the poor performance of Intel MPI when compared with these algorithms, as Intel MPI has 17 times the latency and below 20% of the bandwidth in the most extreme cases. More importantly, MVAPICH2-MIC, with its specific optimizations, could not outperform these algorithms, showing the importance of the algorithms as opposed to focusing exclusively on runtime optimizations.

## 6.7. Impact of Runtime Configuration (pthreads vs. processes) on Xeon Phi

As mentioned before, the Berkeley UPC runtime can implement UPC threads as POSIX threads or as processes. One common problem for communication runtimes is dealing with multiple threads, were internal structures have to be protected by locks. This increases the overhead, that becomes more apparent with a large number of threads. The results obtained in JuRoPA, in Section 5.6, showed better performance over MPI. In Xeon Phi, it looks like the PGAS collectives do not achieve a good performance when compared with MPI in certain cases, like on broadcast. In the experiments done in Stampede, just one process per node was used, with 60 threads per Xeon Phi. However, Figure 6.17 shows clearly that reducing the number of threads per process –and increasing the number of processes per node by the same factor– has a positive performance impact on performance. The performance of the PGAS algorithms when using 15 POSIX threads per process –and 4 processes per node– rivals the performance of the MPI algorithms. Avoiding the use of POSIX threads altogether in the Berkeley UPC runtime is likely to produce even better performance. Nevertheless, this is not possible on Xeon Phi when using a large number of nodes, and using 60 POSIX threads per process is the only way to scale to 15360 cores.



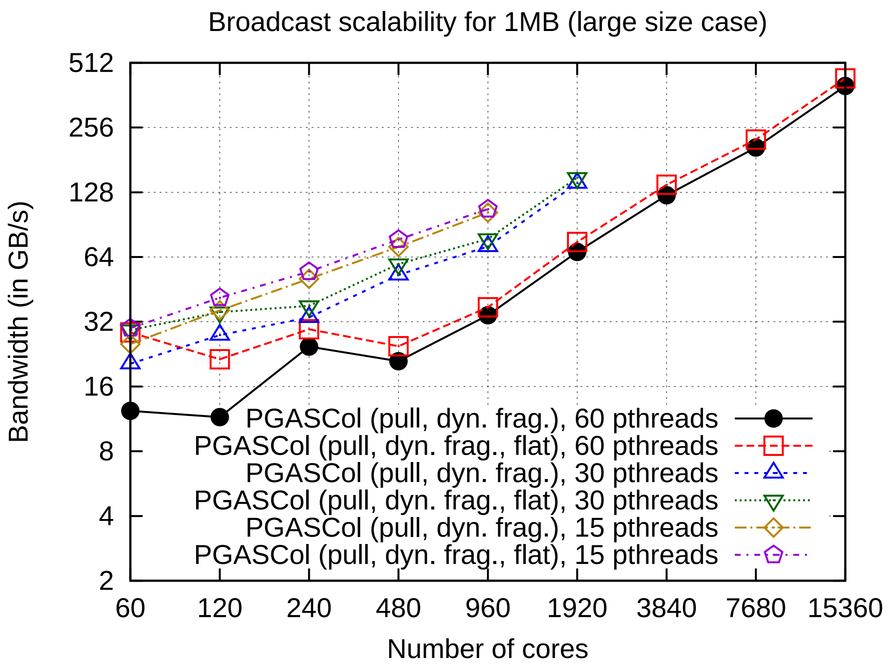


Figure 6.17: Impact of number of pthreads per process on UPC broadcast performance and scalability on Xeon Phi

## 6.8. Conclusions of Chapter 6

This Chapter has analyzed the performance of the proposed PGAS algorithms for broadcast, scatter and gather, on a manycore system, using 2 different processor architectures. The algorithms have been compared with algorithms in leading MPI implementations, tailored specifically to the architecture. The results are favorable to the developed algorithms, and the tests include the larger number of cores used and published in UPC collective experiments.

The analysis of the implementation of the proposed algorithms has shown: (1) as before, PGASCol can outperform in some scenarios the state-of-the-art implementation of their equivalent functions in MPI, bringing another algorithm to the mix, allowing more possibilities for autotuning and choosing the most appropriate algorithm in each situation; (2) in many cases the default MPI algorithm is not the best for Xeon Phi, as the thresholds for switching algorithms seem to have been set with main processors (e.g., Xeon) in mind; (3) the collectives operations overhead is typically much higher on Xeon Phi than on Xeon processors. In some cases it is up to two orders of magnitude higher for the same number of nodes; (4) specific optimizations on the runtime result in measurable benefit, but choosing the correct algorithm plays a major role; (5) better performance could be expected in upcoming UPC Xeon Phi runtimes. It is particularly important the use of multiple processes per node in setups with a large number of cores, minimizing the use of pthreads and therefore minimizing serialization of the access to the communication layer, as well as using specific optimizations for Xeon Phi as seen in MVAPICH2-MIC. Finally, (6) even without specific optimization the proposed algorithms implemented in UPC can outperform the MPI algorithms on Intel MPI and MVAPICH2-MIC in Xeon Phi. Some of the PGAS scatter algorithms outperform the MPI scatter algorithms in bandwidth bound scenarios at large core counts, whereas for gather the performance is better using the PGAS algorithms in both latency and bandwidth bound scenarios.

# Conclusions and Future Work

On the road to Exascale, programming systems with ever-increasing complexity is becoming substantially harder. Managing different levels of the memory hierarchy in shared memory, communication in distributed memory, heterogeneity and other issues such as communication overlapping and vectorization, take most of the time invested in programming scientific applications. Traditional programming models such as message passing might not cope with all these issues and a number of alternatives have been proposed. Among them, PGAS is often profiled as an interesting alternative, due to its programmability and expressiveness. Even though many modern languages focused on high performance computing incorporate PGAS features, just a few of them are being considered by scientist. UPC is one of them, as it is largely based on C, making it familiar and powerful. This Thesis, "*Design of Scalable PGAS Collectives for NUMA and Manycore Systems*" has shown that the PGAS features present in UPC can be effectively used to design and implement efficiently highly scalable collectives that outperform state-of-the-art collective algorithms.

Collective operations are at the core of every communication framework used in high performance computing. The research done on this topic has been extensive, across several years, showing its importance in the supercomputing community. Behind the apparently simple concepts of collective operations lie complex algorithms designed to tackle particular challenges. Optimization of these operations for modern architectures spawns from work on the purely algorithmic level, to reimplementing and adapting different layers of communications runtimes. Despite all the work developed in this arena, there was still missing the combination of some important optimizations, and the benefits of PGAS features on collective libraries on modern hardware had not been sufficiently explored.

The performance of UPC has been analyzed in this Thesis, comparing it to MPI and OpenMP. The analysis of the results obtained in this Thesis has shown that, comparing NPB implementations in a NUMA cluster, MPI is the best performer, mostly due to the use of Fortran vs. C, as Fortran is typically optimized better by the compiler. However, UPC speedups are better for some benchmarks. Moreover, in some benchmarks the performance with 128 cores is not as high as expected, showing that for some workloads the network contention using a high number of cores may be a problem. This will be a bigger issue as the number of cores per node increases in the next years, where a higher network scalability will be required in order to confront this challenge, and highlights the need for topology-aware algorithms. Both MPI and UPC obtain better speedups in shared memory than in the NUMA cluster setup up to 64 cores. However, for 128 cores all the options suffer from remote memory accesses. MPI usually achieves good performance on shared memory, although UPC and OpenMP outperform it in some cases. UPC, due to its expressiveness and ease of programming, is an alternative that has to be taken into account for productive development of parallel applications.

One important contribution of this Thesis is the design and implementation of a complete microbenchmarking suite, which allows, for the first time, assessing communications in UPC. UPC Operations Microbenchmarking Suite (UOMS) is the only microbenchmarking tool for UPC that covers point to point communications, collective communications, awareness of NUMA features, work distribution with `upc_for_all` and read, write and read+write shared memory accesses, and accumulates more than 370 downloads.

The core of this Thesis is the design of a PGAS collective library developed with scalability on modern architectures and portability in mind. Developed from scratch, this library implements a set of collective operations with the following optimizations:

- Appropriate one-sided communication functions, with a push or pull approach depending on the nature of the operation.
- Fixed trees precomputed at initialization time that minimizes the latency of the operations avoiding to compute the tree every time.
- Hierarchical trees carefully mapped to the underlying hardware, minimizing

the use of the slower data paths, and paying particular attention to NUMA architectures.

- Different tree shape in the last level of the hierarchy (binomial and flat).
- Thread binding to ensure proper mapping of the hierarchical trees.
- Efficient pipelining with dynamic and static fragment size, to maximize the bandwidth in medium and large message cases.

The evaluation of the developed library has been carried out in 6 different systems (Stampede, JuRoPA, JUDGE, Finis Terrae, SVG and Superdome), with 5 different processor architectures (Intel Xeon Phi Many Interconnected Core, Intel Xeon Sandy Bridge, Intel Xeon Nehalem, Intel Itanium 2 and AMD Opteron Magny-Cours), and 5 different interconnect technologies (InfiniBand 4x FDR, InfiniBand 4x QDR, InfiniBand 4x DDR, Gigabit Ethernet and Superdome Interconnect). The benefits with regards to state-of-the-art UPC collectives are remarkable in all the setups, as well as with regards to MPI collectives, in multiple scenarios. The analysis of the microbenchmark results of the proposed algorithms has shown that they easily outperform the Berkeley UPC collectives in many cases, despite they being implemented in a lower level (GASNet), and using the same runtime. Moreover, in a large cluster the proposed algorithms always overcome an state-of-the-art MPI implementation, in the scatter and gather operations, whereas it has shown better scalability for the broadcast operation, where they have also outperformed the MPI implementation using a large number of cores. The more important factors to achieve this performance are a tree mapped to the underlying hardware considering all levels, message pipelining and communications overlapping with adequate (pull vs. push) one-sided point-to-point transfers. It should be noted that tree-based collectives are often slightly outperformed by ring algorithms with communication overlapping, in operations where data have to be scattered/gathered from a single point. NUMA binding does not improve significantly the performance in experiments with many nodes, as the network latency becomes dominant. However, with the increasing number of NUMA regions, it is expected that the performance benefits of the developed library will be higher in future systems.

These algorithms have also demonstrated their suitability for manycore systems. Despite the lack of specific optimizations of the runtime for the Xeon Phi platform, their performance has been competitive against the best algorithms of Intel MPI and MVAPICH2-MIC. Moreover, the impossibility of running the Berkeley UPC runtime without relying on pthreads has hindered even further to achieve higher performance. Nevertheless, for gather and scatter operations the proposed algorithms improve their MPI counterparts. Regarding broadcast, even though the performance of the algorithms developed in this Thesis has not improved the performance of the MPI algorithms in Xeon Phi, it has been competitive, and able to scale up to more than 15000 cores, which is the largest evaluation of collectives in manycore architectures up to now. Experiments reducing the number of pthreads as much as possible, and increasing the number of processes accordingly, have also shown that the potential for these algorithms in this platform is even greater than the observed.

The work developed as part of this Thesis has been presented in various conferences and journals. The initial performance evaluations were presented in PGAS [58], EuroPVM/MPI [63] and HPCC [100]. UOMS was presented in ISC [59] and can be downloaded from its website [60]. The proposed algorithms and their evaluation in NUMA systems have been published in the Journal of Cluster Computing [62]. The adaptation of the algorithms to manycore architectures and their evaluation have been submitted for its consideration in the Journal of Parallel Computing [61]. Additionally, an UPC programmability study has been presented in [105], and significant contributions were made to [31], [32] and [104].

Future work on UOMS will further expand its functionality by:

1 Implementing a more efficient and sophisticated mechanism for operating with off-cache data.

2 Providing an option for discarding outliers to avoid their interference in the statistics.

3 Providing an option for changing the root of the collectives in a round robin fashion.

4 Displaying minimum and average bandwidths, in addition to the maximum.

5 Adopting the syntax of the UPC specification 1.3 for the non-blocking memory transfers.

The collective library can be further enhanced by:

1 Including an extra level in the construction of the hierarchical trees, taking into account the switch topology.

2 Shortening initialization times.

3 Autotuning of thresholds for dynamic and static fragmentation/pipelining.

4 Optimizing the reduce local loop, improving its vectorization.

5 Optimizing further the scatter and gather algorithms with non-binomial trees, using multilevel but contiguous trees, allowing the user to provide hints to safely construct these trees.

6 Exploring extra optimizations in operations that do not fit a tree structure.

# Bibliography

[1] B. Amedro, D. Caromel, F. Huet, and V. Bodnartchouk. Java ProActive vs. Fortran MPI: Looking at the Future of Parallel Java. In *Proc. of the 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJCPDC'08)*, pages 1–7, Miami (FL), 2008. `doi:10.1109/IPDPS.2008.4536337`. Cited on page 16.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. `doi:10.1177/109434209100500306`. Cited on page 16.

[3] R. Barriuso and A. Knies. SHMEM user's guide for C. Technical report, Technical report, Cray Research Inc, 1994. Cited on page 7.

[4] R. Brightwell and K. T. Pedretti. Optimizing Multi-Core MPI Collectives with SMARTMAP. In *Proc. of the 3rd International Workshop on Advanced Distributed and Parallel Network Applications (ADPNA'09)*, pages 370–377, Vienna (Austria), 2009. `doi:10.1109/ICPPW.2009.65`. Cited on page 13.

[5] J. M. Bull, J. P. Enright, and N. Ameer. A Microbenchmark Suite for Mixed-Mode OpenMP/MPI. In *Proc. of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism (IWOMP'09)*, pages 118–131, Dresden (Germany), 2009. `doi:10.1007/978-3-642-02303-3_10`. Cited on page 31.

[6] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. A. El-Ghazawi. Performance Monitoring and Evaluation of a UPC Implementation on a

NUMA Architecture. In *Proc. of the 2nd Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'03)*, page 274 (8 pages), Nice (France), 2003. Cited on pages 9 and 16.

[7] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. `doi:10.1177/1094342007078442`. Cited on page 8.

[8] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective Communication on Architectures that Support Simultaneous Communication over Multiple Links. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 2–11, Manhattan (NY), 2006. `doi:10.1145/1122971.1122975`. Cited on page 12.

[9] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proc. of the 4th Conference on Partitioned Global Address Space Programming Model (PGAS'10)*, pages 2:1–2:3, New York (NY), 2010. `doi:10.1145/2020373.2020375`. Cited on page 8.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40(10):519–538, 2005. `doi:10.1145/1103845.1094852`. Cited on page 8.

[11] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou. Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation. In *Proc. of the of 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC'10)*, pages 151–165, Houston, (TX), 2011. `doi:10.1007/978-3-642-19595-2_11`. Cited on page 9.

[12] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran Performance and Potential: An NPB Experimental Study. In *Proc. of the of 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 177–193, College Station (TX), 2003. `doi:10.1007/978-3-540-24644-2_12`. Cited on pages 20 and 26.

[13] Cray. Chapel Language. http://chapel.cray.com. [Last visited: June 2014]. Cited on page 8.

[14] David Koester and Bob Lucas. RandomAccess Benchmark. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/. [Last visited: June 2014]. Cited on page 31.

[15] J. J. Dongarra. The LINPACK Benchmark: An explanation. In *Proc. of the 1st International Conference on Supercomputing (ICS'87)*, pages 456–474, Athens (Greece), 1988. doi:10.1007/3-540-18991-2_27. Cited on page 30.

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990. doi:10.1145/77626.79170. Cited on page 30.

[17] A. Duran, E. Auguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. doi:10.1142/S0129626411000151. Cited on page 6.

[18] T. A. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: a NPB Experimental Study. In *Proc. of the 15th ACM/IEEE Conference on Supercomputing (SC'02)*, pages 1–26, Baltimore (MD), 2002. doi:10.1109/SC.2002.10034. Cited on pages 9, 16, and 18.

[19] T. A. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed. Productivity Analysis of the UPC Language. In *Proc. of the 3rd Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'04)*, pages 1–7, Santa Fe (NM), 2004. Cited on page 8.

[20] T. A. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed. Benchmarking Parallel Compilers: A UPC Case Study. *Future Generation Computer Systems*, 22(7), 2006. doi:10.1016/j.future.2006.02.002. Cited on page 2.

[21] T. A. El-Ghazawi, F. Cantonnet, Y. Yao, and J. Vetter. Evaluation of UPC on the Cray X1. In *Proc. of the 47th Cray User Group meeting (CUG'05)*, page 10 pages, Albuquerque (NM), 2005. Cited on page 16.

[22] T. A. El-Ghazawi and C. Sébastien. UPC Benchmarking Issues. In *Proc. of the 30th IEEE International Conference on Parallel Processing (ICPP'01)*, pages 365–372, Valencia (Spain), 2001. Cited on page 9.

[23] GASPI Project. GASPI Library. `http://www.gaspi.de`. [Last visited: June 2014]. Cited on page 8.

[24] George Washington University. GWU Unified Testing Suite (GUTS). `http://threads.hpcl.gwu.edu/sites/guts`. [Last visited: June 2014]. Cited on pages 29 and 49.

[25] George Washington University High-Performance Computing Laboratory. GWU UPC Benchmarks. `http://www.gwu.edu/~upc/downloads/upc_bench.tar.gz`. [Last visited: June 2014]. Cited on pages 22 and 29.

[26] George Washington University High-Performance Computing Laboratory. GWU UPC SSCA Benchmarks. `http://threads.hpcl.gwu.edu/sites/ssca3`. [Last visited: June 2014]. Cited on page 29.

[27] George Washington University High-Performance Computing Laboratory. UPC NAS Parallel Benchmarks. `http://threads.hpcl.gwu.edu/sites/npb-upc`. [Last visited: June 2014]. Cited on pages 18 and 29.

[28] Y. Gong, B. He, and J. Zhong. Network Performance Aware MPI Collective Communication Operations in the Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2013. `doi:10.1109/TPDS.2013.96`. Cited on page 11.

[29] 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. of the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'12)*, pages 40–47, Madrid (Spain), 2012. `doi:10.1109/ISPA.2012.14`. Cited on page 25.

[30] J. González-Domínguez, M. J. Martín, G. L. Taboada, and J. Touriño. Dense Triangular Solvers on Multicore Clusters using UPC. In *Proc. of the 11th International Conference on Computational Science (ICCS'11)*, pages 231 – 240, Singapore, 2011. `doi:10.1016/j.procs.2011.04.025`. Cited on page 25.

[31] 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. `doi:10.1002/cpe.1914`. Cited on page 156.

[32] 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. In *24th Conference on Supercomputing (SC'11), Research Poster*, Seattle, WA, (USA), 2011. Cited on page 156.

[33] R. L. Graham and G. M. Shipman. MPI Support for Multi-Core Architectures: Optimized Shared Memory Collectives. In *Proc. of the 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'08)*, pages 130–140, Dublin (Ireland), 2008. `doi:10.1007/978-3-540-87475-1_21`. Cited on page 12.

[34] T. Hoefler, C. Siebert, and W. Rehm. A Practically Constant-time MPI Broadcast Algorithm for Large-scale InfiniBand Clusters with Multicast. In *Proc. of the 7th Workshop on Communication Architecture for Clusters (CAC'07)*, pages 1–8, Long Beach (CA), 2007. `doi:10.1109/IPDPS.2007.370475`. Cited on page 13.

[35] P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proc. of the 17th International Conference on Supercomputing (ICS'03)*, pages 63–73, San Francisco (CA), 2003. `doi:10.1145/782814.782825`. Cited on page 9.

[36] IBM Research. X10 Language. `http://x10-lang.org/`. [Last visited: June 2014]. Cited on page 8.

[37] Intel Corporation. Intel MPI Benchmarks. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks`. [Last visited: June 2014]. Cited on page 31.

[38] W. Jalby, C. Lemuet, and X. L. Pasteur. WBTK: a New Set of Microbenchmarks to Explore Memory System Performance for Scientific Computing. *International Journal of High Performance Computing Applications*, 18(2):211–224, 2004. `doi:10.1177/1094342004038945`. Cited on page 31.

[39] N. Jansson. Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-Sided Communication. In *Proc. of the 10th International Meeting on High Performance Computing for Computational Science (VECPAR'12)*, pages 128–139. Kobe, Japan, 2013. `doi:10.1007/978-3-642-38718-0_15`. Cited on page 26.

[40] E. Jeannot and G. Mercier. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In *Proc. of the 16th International European Conference on Parallel and Distributed Computing (Euro-Par'10)*, pages 199–210, Ischia (Italy), 2010. `doi:10.1007/978-3-642-15291-7_20`. Cited on page 118.

[41] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. 2013. Cited on page 7.

[42] W. Jiang, J. Liu, H. wook Jin, D. K. Panda, W. Gropp, and R. Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. In *Proc. of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'04)*, pages 531–538, Chicago (IL), 2004. `doi:10.1109/CCGrid.2004.1336648`. Cited on page 11.

[43] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, NAS-99-011, NASA Ames Research Center, 1999. Cited on page 16.

[44] H. Jin, R. Hood, and P. Mehrotra. A Practical Study of UPC Using the NAS Parallel Benchmarks. In *Proc. of the 3rd Conference on Partitioned Global Address Space Programing Models (PGAS'09)*, pages 8:1–8:7, Ashburn (VA), 2009. `doi:10.1145/1809961.1809973`. Cited on pages 16, 20, and 26.

[45] K. C. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda. Designing Multi-Leader-Based Allgather Algorithms for Multi-Core Clusters. In *Proc. of the 9th Workshop on Communication Architecture for Clusters (CAC'09)*, pages 1–8, Rome (Italy), 2009. `doi:10.1109/IPDPS.2009.5160896`. Cited on pages 13 and 49.

[46] K. C. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with Scatter and Gather. In *Proc. of the 10th Workshop on Communication Architecture for Clusters (CAC'10)*, pages 1–8, Atlanta (GA), 2010. `doi:10.1109/IPDPSW.2010.5470853`. Cited on pages 11 and 49.

[47] K. C. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D. K. Panda. Designing Optimized MPI Broadcast and Allreduce for Many Integrated Core (MIC) InfiniBand Clusters. In *Proc. of the 21st Annual Symposium on High-Performance Interconnects (HOTI'13)*, San Jose (CA), 2013. `doi:10.1109/HOTI.2013.26`. Cited on page 12.

[48] M. J. Koop, J. K. Sridhar, and D. K. Panda. Scalable MPI Design over InfiniBand using eXtended Reliable Connection. In *Proc. of the 10th IEEE International Conference on Cluster Computing (Cluster'08)*, pages 203–212, Tsukuba (Japan), 2008. `doi:10.1109/CLUSTR.2008.4663773`. Cited on page 57.

[49] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*, pages 81–92, San Diego (CA), 2003. `doi:10.1109/MICRO.2003.1253185`. Cited on page 5.

[50] R. Kumar, A. R. Mamidala, and D. K. Panda. Scaling Alltoall Collective on Multi-Core Systems. In *Proc. of the 8th Workshop on Communication Architecture for Clusters (CAC'08)*, pages 1–8, Miami (FL), 2008. `doi:10.1109/IPDPS.2008.4536141`. Cited on pages 12 and 49.

[51] Lawrence Berkeley National Laboratory and University of California, Berkeley. Berkeley UPC. `http://upc.lbl.gov`. [Last visited: June 2014]. Cited on pages 9, 11, and 41.

[52] Lawrence Livermore National Laboratory. Interleaved or Random Benchmark (IOR). `https://github.com/chaos/ior`. [Last visited: June 2014]. Cited on page 31.

[53] S. Li, T. Hoefler, and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. In *Proc. of the 22nd International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'13)*, pages 85–96, New York (NY), 2013. `doi:10.1145/2462902.2462903`. Cited on page 13.

[54] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. Panda, and P. Wyckoff. Microbenchmark Performance Comparison of High-Speed Cluster Interconnects. *IEEE Micro*, 24(1):42–51, Jan 2004. `doi:10.1109/MM.2004.1268994`. Cited on page 31.

[55] J. A. Lorenzo, F. F. Rivera, P. Tuma, and J. C. Pichel. On the Influence of Thread Allocation for Irregular Codes in NUMA Systems. In *Proc. of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, pages 146–153, Hiroshima, (Japan), 2009. `doi:10.1109/PDCAT.2009.42`. Cited on page 51.

[56] M. Luo, J. Jose, S. Sur, and D. K. Panda. Multi-Threaded UPC Runtime with Network Endpoints: Design Alternatives and Evaluation on Multi-Core Architectures. In *Proc. of the 18th International Conference on High Performance Computing (HiPC'11)*, pages 1–10, 2011. `doi:10.1109/HiPC.2011.6152734`. Cited on page 9.

[57] M. Luo, M. Li, A. Venkatesh, X. Lu, and D. K. Panda. UPC on MIC: Early Experiences with Native and Symmetric Modes. In *Proc. of the 7th International Conference on PGAS Programming Models (PGAS'13)*, Edinburgh (UK), 2013. Cited on page 14.

[58] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan. UPC Performance Evaluation on

a Multicore System. In *Proc. of the 3rd Conference on Partitioned Global Address Space Programing Models (PGAS'09)*, pages 9:1–9:7, Ashburn (VA), 2009. `doi:10.1145/1809961.1809974`. Cited on pages xxxi and 156.

[59] D. A. Mallón, J. C. Mouriño, A. Gómez, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan. UPC Operations Microbenchmarking Suite. In *25th International Supercomputing Conference (ISC'10), Research Poster*, Hamburg (Germany), 2010. Cited on pages xxxi and 156.

[60] D. A. Mallón and G. L. Taboada. UOMS Webpage. `http://upc.cesga.es`. [Last visited: June 2014]. Cited on pages xxxi, 39, 78, and 156.

[61] D. A. Mallón, G. L. Taboada, and L. Koesterke. MPI and UPC Broadcast, Scatter and Gather Algorithms in Xeon Phi. *Submitted to Journal of Parallel Computing*, pages 1–25, 2014. Cited on pages xxxi and 156.

[62] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. González-Domínguez, A. Gómez, and B. Wibecan. Scalable PGAS Collective Operations in NUMA Clusters. *Journal of Cluster Computing*, pages 1–23, 2014. In press. Cited on pages xxxi and 156.

[63] 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. of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, pages 174–184, Espoo (Finland), 2009. `doi:10.1007/978-3-642-03770-2_24`. Cited on pages xxxi and 156.

[64] D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message Passing in Java. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'09)*, pages 181–190, Weimar (Germany), 2009. `doi:10.1109/PDP.2009.59`. Cited on pages 16 and 17.

[65] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *Proc. of the 8th IEEE/ACM International Symposium on*

*Cluster Computing and the Grid (CCGrid'08)*, pages 130–137, Lyon (France), 2008. `doi:10.1109/CCGRID.2008.87`. Cited on page 13.

[66] Matteo Frigo and Steven G. Johnson. benchFFT. `http://www.fftw.org/benchfft`. [Last visited: June 2014]. Cited on page 30.

[67] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995. Cited on page 30.

[68] J. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991. `doi:10.1145/103727.103729`. Cited on page 68.

[69] Q. Miao, G. Sun, J. Shan, and G. Chen. Single Data Copying for MPI Communication Optimization on Shared Memory System. In *Proc. of the 7th International Conference on Computational Science (ICCS'07)*, pages 700–707, Beijing (China), 2007. `doi:10.1007/978-3-540-72584-8_93`. Cited on page 13.

[70] Michigan Tech. Collectives Reference Implementation. `http://www.upc.mtu.edu/collectives/col1.html`. [Last visited: June 2014]. Cited on pages 41, 78, and 94.

[71] J. C. Mouriño, A. Gómez, J. M. Taboada, L. Landesa, J. M. Bértolo, F. Obelleiro, and J. L. Rodríguez. High Scalability Multipole Method. Solving Half Billion of Unknowns. *Computer Science - R&D*, 23(3–4):169–175, 2009. `doi:10.1007/s00450-009-0075-7`. Cited on page 48.

[72] G. Mozdzynski, M. Hamrud, N. Wedi, J. Doleschal, and H. Richardson. A PGAS Implementation by Co-design of the ECMWF Integrated Forecasting System (IFS). In *Proc. of the 24th Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 652–661, Salt Lake City (UT), 2012. `doi:10.1109/SC.Companion.2012.90`. Cited on page 15.

[73] NASA Advanced Computing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`. [Last visited: June 2014]. Cited on page 16.

[74] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008. `doi:10.1145/1365490.1365500`. Cited on page 6.

[75] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers. In *Proc. of the 7th Conference of Supercomputing (SC'94)*, pages 340–349, 816, Washington (D.C.), Nov 1994. `doi:10.1109/SUPERC.1994.344297`. Cited on page 7.

[76] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006. `doi:10.1177/1094342006064503`. Cited on page 7.

[77] R. Nishtala and K. A. Yelick. Optimizing Collective Communication on Multicores. In *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, pages 1–6, Berkeley (CA), 2009. Cited on page 12.

[78] R. Nishtala, Y. Zheng, P. H. Hargrove, and K. A. Yelick. Tuning Collective Communication for Partitioned Global Address Space Programming Models. *Journal of Parallel Computing*, (37):576–591, 2011. `doi:10.1016/j.parco.2011.05.006`. Cited on page 13.

[79] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998. `doi:10.1145/289918.289920`. Cited on page 7.

[80] R. W. Numrich and J. Reid. Co-Arrays in the Next Fortran Standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005. `doi:10.1145/1080399.1080400`. Cited on page 7.

[81] Ohio State University. OSU Microbenchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`. [Last visited: June 2014]. Cited on page 31.

[82] Ohio State University. OSU Microbenchmarks Changelog. `http://mvapich.cse.ohio-state.edu/svn/mpi-benchmarks/branches/4.3/CHANGES`. [Last visited: June 2014]. Cited on page 39.

[83] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`. [Last visited: June 2014]. Cited on page 6.

[84] Oracle Labs. Fortress Language. `https://projectfortress.java.net`. [Last visited: June 2014]. Cited on page 8.

[85] Pacific Northwest National Laboratory. Global Arrays Webpage. `http://hpc.pnl.gov/globalarrays/`. [Last visited: June 2014]. Cited on page 7.

[86] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, and D. K. Panda. MVAPICH-PRISM: A Proxy-based Communication Framework Using InfiniBand and SCIF for Intel MIC Clusters. In *Proc. of the 25th International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, Denver (CO), 2013. `doi:10.1145/2503210.2503288`. Cited on page 12.

[87] S. Potluri, A. Venkatesh, D. Bureddy, K. Kandalla, and D. K. Panda. Efficient Intra-node Communication on Intel-MIC Clusters. In *Proc. of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13)*, pages 128–135, Delft (The Netherlands), 2013. `doi:10.1109/CCGrid.2013.86`. Cited on page 14.

[88] Y. Qian. *Design and Evaluation of Efficient Collective Communications on Modern Interconnects and Multi-Core Clusters*. PhD thesis, Electrical & Computer Engineering – Queen's University, 2010. Cited on page 13.

[89] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proc. of the 17th*

*Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'09)*, pages 427–436, Weimar (Germany), 2009. doi:10.1109/PDP.2009.43. Cited on page 6.

[90] R. Rabenseifner, G. Hager, G. Jost, and R. Keller. Hybrid MPI and OpenMP Parallel Programming. In *Proc. of the 13th European PVM/MPI Users Group Meeting (EuroPVM/MPI'06)*, page 11, Bonn (Germany), 2006. doi:10.1007/11846802_10. Cited on page 6.

[91] R. Rabenseifner and A. E. Koniges. Effective File-I/O Bandwidth Benchmark. In *Proc. of the 6th International Euro-Par Conference on Parallel Processing (Euro-Par'00)*, pages 1273–1283, Munich (Germany), 2000. doi:10.1007/3-540-44520-X_179. Cited on page 31.

[92] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the Case for an ARM-based HPC System. *Future Generation Computer Systems*, 36, 2013. doi:10.1016/j.future.2013.07.013. Cited on page 1.

[93] A. Ramachandran, J. Vienne, R. Van Der Wijngaart, L. Koesterke, and I. Sharapov. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *Proc. of the 42nd International Conference on Parallel Processing (ICPP'13)*, pages 736–743, Lyon (France), 2013. doi:10.1109/ICPP.2013.87. Cited on page 16.

[94] S. Ramos and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems: a Case-Study with Xeon Phi. In *Proc. of the 22nd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'13)*, pages 97–108, New York (NY), 2013. doi:10.1145/2462902.2462916. Cited on page 12.

[95] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proc. of the 8th IEEE International Symposium on Workload Characterization (IISWC'11)*, pages 137–148, Austin (TX), 2011. doi:10.1109/IISWC.2011.6114174. Cited on page 16.

[96] O. Serres, A. Anbar, S. Merchant, and T. El-Ghazawi. Experiences with UPC on TILE-64 processor. In *IEEE Aerospace Conference (AeroConf'11)*, pages 1–9, Big Sky (MT), 2011. `doi:10.1109/AERO.2011.5747452`. Cited on page 9.

[97] G. M. Shipman, S. Poole, P. Shamis, and I. Rabinovitz. X-SRQ - Improving Scalability and Performance of Multi-Core InfiniBand Clusters. In *Proc. of the 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'08)*, pages 33–42, Dublin (Ireland), 2008. `doi:10.1007/978-3-540-87475-1_11`. Cited on page 57.

[98] G. Steele. Fortress: A New Programming Language for Scientific Computing. *Sun Labs Open House*, 2005. Cited on page 8.

[99] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010. `doi:10.1109/MCSE.2010.69`. Cited on page 6.

[100] G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, J. C. Mouriño, D. A. Mallón, and A. Gómez. Performance Evaluation of Unified Parallel C Collective Communications. In *Proc. of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC'09)*, pages 69–78, Seoul (Korea), 2009. `doi:10.1109/HPCC.2009.88`. Cited on pages xxx, 41, and 156.

[101] C. Teijeiro, G. Sutmann, G. L. Taboada, and J. Touriño. Parallel Brownian Dynamics Simulations with the Message-Passing and PGAS Programming Models. *Computer Physics Communications*, 184(4):1191–1202, 2013. `doi:10.1016/j.cpc.2012.12.015`. Cited on page 26.

[102] C. Teijeiro, G. Sutmann, G. L. Taboada, and J. Touriño. Parallel Simulation of Brownian Dynamics on Shared Memory Systems with OpenMP and Unified Parallel C. *The Journal of Supercomputing*, 65(3):1050–1062, 2013. `doi:10.1007/s11227-012-0843-1`. Cited on page 26.

[103] C. Teijeiro, G. Taboada, J. Touriño, and R. Doallo. Design and Implementation of MapReduce Using the PGAS Programming Model with UPC.

In *Proc. of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS'11)*, pages 196–203, Tainan (Taiwan), 2011. `doi:10.1109/ICPADS.2011.162`. Cited on page 26.

[104] C. Teijeiro, G. L. Taboada, J. Touriño, R. Doallo, J. Mouriño, D. Mallón, and B. Wibecan. Design and Implementation of an Extended Collectives Library for Unified Parallel C. *Journal of Computer Science and Technology*, 28(1):72–89, 2013. `doi:10.1007/s11390-013-1313-9`. Cited on page 156.

[105] 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. of the 3rd Conference on Partitioned Global Address Space Programing Models (PGAS'09)*, pages 10:1–10:7, Ashburn (VA), 2009. `doi:10.1145/1809961.1809975`. Cited on pages 8 and 156.

[106] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. `doi:10.1177/1094342005051521`. Cited on page 13.

[107] Top500.org. Finis Terrae Supercomputer at TOP500 List. `http://www.top500.org/system/9500`. [Last visited: June 2014]. Cited on pages 16 and 73.

[108] Top500.org. JUDGE Supercomputer at TOP500 List. `http://www.top500.org/system/177621`. [Last visited: June 2014]. Cited on page 75.

[109] Top500.org. JuRoPA Supercomputer at TOP500 List. `http://www.top500.org/system/176488`. [Last visited: June 2014]. Cited on page 76.

[110] Top500.org. Stampede Supercomputer at TOP500 List. `http://top500.org/system/177931`. [Last visited: June 2014]. Cited on pages 123 and 125.

[111] Top500.org. TOP500 List. `http://www.top500.org`. [Last visited: June 2014]. Cited on page 30.

[112] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *Proc. of the 8th Workshop on*

*Communication Architecture for Clusters (CAC'08)*, pages 1–7, Miami (FL), 2008. doi:10.1109/IPDPS.2008.4536139. Cited on page 13.

[113] B. Tu, J. Fan, J. Zhan, and X. Zhao. Performance Analysis and Optimization of MPI Collective Operations on Multi-Core Clusters. *Journal of Supercomputing*, 60(1):141–162, 2012. doi:10.1007/s11227-009-0296-3. Cited on pages 12 and 49.

[114] UPC Consortium. UPC Language. https://upc-lang.org. [Last visited: June 2014]. Cited on page 7.

[115] UPC Consortium. UPC Language Specifications Version 1.3. https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf. [Last visited: June 2014]. Cited on page 11.

[116] UPC Consortium. UPC Optional Library Specifications Version 1.3. https://upc-lang.org/assets/Uploads/spec/upc-lib-optional-spec-1.3.pdf. Cited on page 11.

[117] UPC Consortium. UPC Required Library Specifications Version 1.3. https://upc-lang.org/assets/Uploads/spec/upc-lib-required-spec-1.3.pdf. [Last visited: June 2014]. Cited on page 11.

[118] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Automatically Tuned Collective Communications. In *Proc. of the 13th ACM/IEEE Conference on Supercomputing (SC'00)*, pages 1–11, Dallas (TX), 2000. doi:10.1109/SC.2000.10024. Cited on page 13.

[119] M. K. Velamati, A. Kumar, N. Jayam, G. Senthilkumar, P. K. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan. Optimization of Collective Communication in Intra-cell MPI. In *Proc. of the 14th International Conference on High Performance Computing (HiPC'07)*, pages 488–499, Goa (India), 2007. doi:10.1007/978-3-540-77220-0_45. Cited on page 13.

[120] B. Wibecan. Proposal for Privatizability Functions. http://www2.hpcl.gwu.edu/pgas09/HP_UPC_Proposal.pdf, 2009. [Last visited: June 2014]. Cited on page 50.

[121] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998. doi:10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H. Cited on page 7.

[122] Z. Ryne and S. Seidel. Ideas and Specifications for the New One-sided Collective Operations in UPC. http://www.upc.mtu.edu/papers/OnesidedColl.pdf. [Last visited: June 2014]. Cited on page 11.

# Appendix: UOMS User Manual

***CESGA Alliance***

# UPC Operations Microbenchmarking Suite 1.1 User's manual

*Authors:*
PhD. Guillermo López Taboada[1]
Damián Álvarez Mallón[2]

[1]taboada@udc.es
[2]dalvarez@cesga.es

## Contents

## 1 Contact

You can contact us at:

Galicia Supercomputing Center (CESGA)
`http://www.cesga.es`
Santiago de Compostela, Spain
upc@cesga.es

PhD. Guillermo Lopez Taboada
Computer Architecture Group (CAG)
`http://gac.des.udc.es/index_en.html`
University of A Coruña, Spain
taboada@udc.es

## 2 Acknowledgments

This work was funded by Hewlett-Packard Spain and partially supported by the Ministry of Science and Innovation of Spain under Project TIN2007-67537-C03-02 and by the Galician Government (Xunta de Galicia, Spain) under the Consolidation Program of Competitive Research Groups (Ref. 3/2006 DOGA 12/13/2006). We gratefully thank Brian Wibecan for his comments and for share with us his thoughts and knowledge. Also, we thank Jim Bovay for his support, and CESGA, for providing access to the FinisTerrae supercomputer.

## 3 Files in this benchmarking suite

- `doc/manual.pdf`: This file. User's manual.
- `COPYING and COPYING.LESSER`: Files containing the use and redistribution terms (license).
- `changelog.txt`: File with changes in each release.
- `Makefile`: Makefile to build the benchmarking suite. It relies on the src/Makefile file.
- `src/affinity.upc`: UPC code with affinity-related tests.
- `src/config/make.def.template.*`: Makefile templates for HP UPC and Berkeley UPC.
- `src/config/parameters.h`: Header with some customizable parameters.
- `src/defines.h`: Header with needed definitions.
- `src/headers.h`: Header with HUCB functions headers.
- `src/mem_manager.upc`: Memory-related functions for allocation and freeing.

- `src/UOMS.upc`: Main file. It contains the actual benchmarking code.
- `src/init.upc`: Code to initialize some structures and variables.
- `src/Makefile`: Makefile to build the benchmarking suite.
- `src/timers/timers.c`: Timing functions.
- `src/timers/timers.h`: Timing functions headers.
- `src/utils/data_print.upc`: Functions to output the results.
- `src/utils/utilities.c`: Auxiliary functions.

## 4 Operations tested

- `upc_forall` (read elements of a shared array)
- `upc_forall` (write elements of a shared array)
- `upc_forall` (read+write elements of a shared array)
- `for` (read elements of a shared array)
- `for` (write elements of a shared array)
- `for` (read+write elements of a shared array)
- `upc_barrier`
- `upc_all_broadcast`
- `upc_all_scatter`
- `upc_all_gather`
- `upc_all_gather_all`
- `upc_all_permute`
- `upc_all_exchange`
- `upc_all_reduceC`
- `upc_all_prefix_reduceC`
- `upc_all_reduceUC`
- `upc_all_prefix_reduceUC`
- `upc_all_reduceS`
- `upc_all_prefix_reduceS`

- upc_all_reduceUS
- upc_all_prefix_reduceUS
- upc_all_reduceI
- upc_all_prefix_reduceI
- upc_all_reduceUI
- upc_all_prefix_reduceUI
- upc_all_reduceL
- upc_all_prefix_reduceL
- upc_all_reduceUL
- upc_all_prefix_reduceUL
- upc_all_reduceF
- upc_all_prefix_reduceF
- upc_all_reduceD
- upc_all_prefix_reduceD
- upc_all_reduceLD
- upc_all_prefix_reduceLD
- upc_memcpy (remote)
- upc_memget (remote)
- upc_memput (remote)
- upc_memcpy (local)
- upc_memget (local)
- upc_memput (local)
- memcpy (local)
- memmove (local)
- upc_memcpy_async (remote)
- upc_memget_async (remote)
- upc_memput_async (remote)
- upc_memcpy_async (local)

- `upc_memget_async` (local)
- `upc_memput_async` (local)
- `upc_memcpy_asynci` (remote)
- `upc_memget_asynci` (remote)
- `upc_memput_asynci` (remote)
- `upc_memcpy_asynci` (local)
- `upc_memget_asynci` (local)
- `upc_memput_asynci` (local)
- `upc_all_alloc`
- `upc_free`

The `upc_forall` and `for` benchmarks test the performance of accesses to a shared `int` array in read, write and read+write operations. The `upc_forall` benchmark distributes the whole workload across threads, whereas in the `for` benchmark all the work is performed by thread 0. This is useful for testing the speed of remote accesses and optimization techniques such as coalescing. The operation performed in read is a sum of a variable in the stack and the current element in the array, to prevent the compiler from dropping the first $N-1$ iterations. The operation performed in write is a simply update of the elements with its position in the array. The operation performed in read+write is a sum of the current element and its position in the array.

In bulk memory transfer operations there are two modes: remote and local. Remote mode will copy data from one thread to another, whereas local mode, will copy data from one thread to another memory region with affinity to the same thread.

## 5 Customizable parameters

### 5.1 Compile time

In the `src/config/parameters.h` file you can customize some parameters at compile time. They are:

- `NUMCORES`: If defined it will override the detection of the number of cores. If not defined the number of cores is set through the `sysconf(_SC_NPROCESSORS_ONLN)` system call.
- `ASYNC_MEM_TEST`: If defined asynchronous memory transfer tests will be built. Default is defined.
- `ASYNCI_MEM_TEST`: If defined asynchronous memory transfer with implicit handlers tests will be built. Default is defined.
- `MINSIZE`: The minimum message size to be used in the benchmarking. Default is 4 bytes.
- `MAXSIZE`: The maximum message size to be used in the benchmarking. Default is 16 megabytes.

## 5.2 Run time

The following flags can be used at run time in the command line:

- `-help`: Print usage information and exits.
- `-version`: Print UOMS version and exits.
- `-off_cache`: Enable cache invalidation. Be aware that the cache invalidation greatly increases the memory consumption. Also, note that for block sizes smaller than the cache line size it will not have any effect.
- `-warmup`: Enable a warmup iteration.
- `-reduce_op OP`: Choose the reduce operation to be performed by `upc_all_reduceD` and `upc_all_prefix_reduceD`. Valid operations are:
  - `UPC_ADD (default)`
  - `UPC_MULT`
  - `UPC_LOGAND`
  - `UPC_LOGOR`
  - `UPC_AND`
  - `UPC_OR`
  - `UPC_XOR`
  - `UPC_MIN`
  - `UPC_MAX`
- `-sync_mode MODE`: Choose the synchronization mode for the collective operations. Valid modes are:
  - `UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC (default)`
  - `UPC_IN_ALLSYNC|UPC_OUT_MYSYNC`
  - `UPC_IN_ALLSYNC|UPC_OUT_NOSYNC`
  - `UPC_IN_MYSYNC|UPC_OUT_ALLSYNC`
  - `UPC_IN_MYSYNC|UPC_OUT_MYSYNC`
  - `UPC_IN_MYSYNC|UPC_OUT_NOSYNC`
  - `UPC_IN_NOSYNC|UPC_OUT_ALLSYNC`
  - `UPC_IN_NOSYNC|UPC_OUT_MYSYNC`
  - `UPC_IN_NOSYNC|UPC_OUT_NOSYNC`
- `-msglen FILE`: Read user defined problem sizes from `FILE` (in bytes). If specified it will override `-minsize` and `-maxsize`
- `-minsize SIZE`: Specifies the minimum block size (in bytes). Sizes will increase by a factor of 2

- `-maxsize SIZE`: Specifies the maximum block size (in bytes)
- `-time SECONDS`: Specifies the maximum run time in seconds for each block size. Disabled by default. Important: this setting will not interrupt an ongoing operation
- `-input FILE`: Read user defined list of benchmarks to run from `FILE`. Valid benchmark names are:
  - `upc_forall_read`
  - `upc_forall_write`
  - `upc_forall_readwrite`
  - `for_read`
  - `for_write`
  - `for_readwrite`
  - `upc_barrier`
  - `upc_all_broadcast`
  - `upc_all_scatter`
  - `upc_all_gather`
  - `upc_all_gather_all`
  - `upc_all_exchange`
  - `upc_all_permute`
  - `upc_memget`
  - `upc_memput`
  - `upc_memcpy`
  - `local_upc_memget`
  - `local_upc_memput`
  - `local_upc_memcpy`
  - `memcpy`
  - `memmove`
  - `upc_all_alloc`
  - `upc_free`
  - `upc_all_reduceC`
  - `upc_all_prefix_reduceC`
  - `upc_all_reduceUC`
  - `upc_all_prefix_reduceUC`
  - `upc_all_reduceS`
  - `upc_all_prefix_reduceS`
  - `upc_all_reduceUS`

- upc_all_prefix_reduceUS
- upc_all_reduceI
- upc_all_prefix_reduceI
- upc_all_reduceUI
- upc_all_prefix_reduceUI
- upc_all_reduceL
- upc_all_prefix_reduceL
- upc_all_reduceUL
- upc_all_prefix_reduceUL
- upc_all_reduceF
- upc_all_prefix_reduceF
- upc_all_reduceD
- upc_all_prefix_reduceD
- upc_all_reduceLD
- upc_all_prefix_reduceLD
- upc_memget_async
- upc_memput_async
- upc_memcpy_async
- local_upc_memget_async
- local_upc_memput_async
- local_upc_memcpy_async
- upc_memget_asynci
- upc_memput_asynci
- upc_memcpy_asynci
- local_upc_memget_asynci
- local_upc_memput_asynci
- local_upc_memcpy_asynci

## 6 Compilation

To compile the suite you have to setup a correct `src/config/make.def` file. Templates are provided to this purpose. The needed parameters are:

- `CC`: Defines the C compiler used to compile the C code. Please note this has nothing to do with the resulting C code generated from the UPC code if your UPC compiler is a source to source compiler.

- `CFLAGS`: Defines the C flags used to compile the C code. Please note this has nothing to do with the resulting C code generated from the UPC code if your UPC compiler is a source to source compiler
- `UPCC`: Defines the UPC compiler used to compile the suite
- `UPCFLAGS`: Defines the UPC compiler flags used to compile the suite. Please note you should not specify the number of threads flag at this point
- `UPCLINK`: Defines the UPC linker used to link the suite
- `UPCLINKFLAGS`: Defines the UPC linker flags used to link the suite
- `THREADS_SWITCH`: Defines the correct switch to set the desired number of threads. It is compiler dependant, and also includes any blank space after the switch

Once you have set up your `make.def` file you can compile the suite.

For a static thread setup type:
`make NTHREADS=NUMBER_OF_UPC_THREADS`
E.g., for 128 threads:
`make NTHREADS=128`

For a dynamic thread setup just type:
`make`

## 7 Timers used

This suite uses high-resolution timers in IA64 architecture. In particular it uses the Interval Timer Counter (`AR.ITC`). For other architectures it uses the `hpupc_ticks_now` if you are using HP UPC, or `bupc_ticks_now` if you are using Berkeley UPC, whose precision depends on the specific architecture. If none of this requirements are met the suite uses the default `gettimeofday` function. However, the granularity of this function only allows to measure microseconds, rather than nanoseconds.

## 8 Output explanation

This is an output example of the broadcast:

```
#---------------------------------------------------
# Benchmarking upc_all_broadcast
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions  t_min[nsec]  t_max[nsec]    t_avg[nsec] BW_aggregated[MB/sec]
            4           20        19942     48820275     2463315.85                  0.00
            8           20        19942        22922       21457.25                  0.70
           16           20        19942        22397       21420.10                  1.43
           32           20        19942        22235       21626.35                  2.88
           64           20        20277        33610       22886.00                  3.81
```

```
          128           20        20285        22812     21676.60              11.22
          256           20        20767        22845     22230.50              22.41
          512           20        20767        23020     22314.85              44.48
         1024           20        22777        29255     24169.85              70.01
         2048           20        23705        25425     24603.85             161.10
         4096           20        24562        27097     26437.60             302.32
         8192           20        29885        33205     32174.35             493.42
        16384           20        42492        44735     43919.35             732.49
        32768           10        68317        70052     69490.00             935.53
        65536           10       121610       123837    122635.00            1058.42
       131072           10       227550       231515    229323.50            1132.30
       262144           10       437645       444740    441354.00            1178.86
       524288           10       861287       871700    867619.70            1202.91
      1048576            5      1702722      1704420   1703642.40            1230.42
      2097152            5      3417170      3435637   3429128.40            1220.82
      4194304            5      6830267      6839535   6834224.40            1226.49
      8388608            2     13434382     13469047  13451715.00            1245.61
     16777216            2     27310152     27343357  27326755.00            1227.15
     33554432            1     54294385     54294385  54294385.00            1236.02
```

The header indicates the benchmarked function and the number of processes involved. The first column shows the block size used for each particular row. The second column is the number of repetitions performed for that particular message size. The following three columns are, respectively, the minimum, maximum and average latencies. The last column shows the aggregated bandwidth calculated using the maximum latencies. Therefore, the bandwidth reported is the minimum bandwidth achieved in all the repetitions.

Moreover, when 2 threads are used, affinity tests are performed. This way you can measure the effects of data locality in NUMA systems, if the 2 threads run in the same machine. This feature may be useful even when the 2 threads run in different machines. E.g.: Machines with non-uniform access to the network interface, like quad-socket Opteron/Nehalem-based machines, or cell-based machines like HP Integrity servers. The output of this tests is preceded with something like:

```
#-----------------------------------------------------------
# using #cores = 0 and 1 (Number of cores per node: 16)
# CPU Mask: 1000000000000000 (core 0), 0100000000000000 (core 1)
#-----------------------------------------------------------
```

All tests after these lines are performed using core 0 (thread 0) and core 1 (thread 1) until another affinity header is showed.