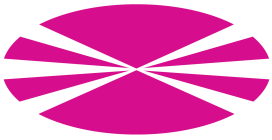
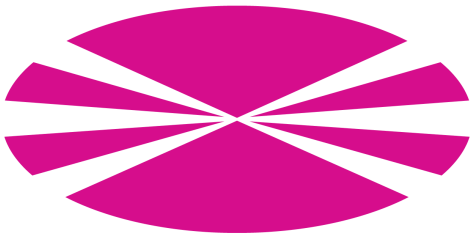
Extended Collectives Library for Unified Parallel C

Carlos Teijeiro Barjas



Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems
University of A Coruña, Spain



PhD Thesis

Extended Collectives Library for Unified Parallel C

Carlos Teijeiro Barjas

January 2013

PhD Advisors:
Guillermo López Taboada
Juan Touriño Domínguez

Dr. Guillermo López Taboada
Profesor Contratado Doutor
Dpto. de Electrónica e Sistemas
Universidade da Coruña

Dr. Juan Touriño Domínguez
Catedrático de Universidade
Dpto. de Electrónica e Sistemas
Universidade da Coruña

CERTIFICAN

Que a memoria titulada "*Extended Collectives Library for Unified Parallel C*" foi realizada por D. Carlos Teijeiro Barjas sob a nosa dirección no Departamento de Electrónica e Sistemas da Universidade da Coruña e conclúe a Tese de Doutoramento que presenta para a obtención do título de Doutor pola Universidade da Coruña coa Mención de Doutor Internacional.

Na Coruña, o 24 de Xaneiro de 2013

Asdo.: Guillermo López Taboada
Director da Tese de Doutoramento

Asdo.: Juan Touriño Domínguez
Director da Tese de Doutoramento

Asdo.: Carlos Teijeiro Barjas
Autor da Tese de Doutoramento

Resumo da Tese de Doutoramento

Introdución

Na actualidade, o desenvolvemento de novas arquitecturas de computadores híbridas de memoria compartida/distribuída que fan uso de nodos de procesadores multinúcleo, coa posible asistencia de aceleradores hardware tales como procesadores gráficos (GPUs), está a proporcionar unha maior capacidade de procesamento aos sistemas computacionais paralelos e heteroxéneos baseados nelas. Consecuentemente, o desenvolvemento de aplicacións para computación de altas prestacións nestes sistemas requere da explotación eficiente dos seus recursos, de xeito que se optimicen, entre outros factores, a explotación da programación multifío e o acceso á memoria para cada un deles. Porén, a crecente complexidade das arquitecturas paralelas actuais leva consigo un incremento na dificultade da súa programación, requerendo un maior esforzo de programación por parte dos desenvolvedores de aplicacións. As linguaxes e bibliotecas de programación paralela utilizadas tradicionalmente, tanto en memoria compartida (OpenMP) como en memoria distribuída (MPI), permiten obter solucións que resultan ser ou ben ineficientes e pouco escalables por falta de flexibilidade na súa definición, ou ben complexas para a obtención dun rendemento adecuado.

Esta situación fai medrar o interese pola obtención de novos paradigmas de programación que favorezan unha maior produtividade dos programadores de aplicacións paralelas, na procura dunha mellor integración da arquitectura do computador híbrido no xeito de programar para a súa explotación eficiente. Entre as diferentes aproximacións, o paradigma PGAS (Partitioned Global Address Space, ou espazo de direccións globais particionado) acadou un desenvolvemento relevante. Este modelo

de programación establece unha visión da memoria dividida en dous espazos: partillado (visible e accesible para todos os fíos de execución - *threads* - do programa) e privado (bloques de acceso local exclusivo para cada un dos *threads*), co obxectivo de ofrecer un ambiente máis adaptado a arquitecturas híbridas de memoria compartida/distribuída que simplifique a súa programación.

O paradigma PGAS é empregado como modelo de memoria na linguaxe Unified Parallel C (UPC), que é unha extensión paralela da linguaxe C. UPC inclúe construcións específicas de apoio ao desenvolvemento de aplicacións paralelas, como (1) movementos de datos remotos mediante asignacións a variables compartidas, (2) constantes predefinidas que definen parámetros do programa paralelo, como o número de *threads*, e (3) bibliotecas de funcións para movemento de datos en memoria e operacións colectivas. Con todo, as especificacións estándar da linguaxe UPC son relativamente recentes, e hai procesos de debate abertos dentro da comunidade de usuarios e programadores de UPC para a súa mellora e ampliación. Do mesmo xeito, diversos centros de investigación e fabricantes de software desenvolveron compiladores e ambientes de execución para a linguaxe UPC, que ofrecen soporte para as funcionalidades estándar da linguaxe, alén de distintos conxuntos de códigos de probas (*benchmarks*) para a avaliación do seu rendemento.

Porén, a adecuación de UPC para o seu uso en aplicacións en computación de altas prestacións aínda debe ser confirmada por un maior grao de aceptación e introdución na comunidade científica. Na actualidade, UPC aínda carece dunha implantación maioritaria neste ámbito, debido a diversos factores. A posición preponderante de linguaxes de programación paralela tradicionalmente usadas (MPI, OpenMP) fai difícil a súa introdución, pola incidencia da curva de aprendizaxe en programadores xa expertos nese eido. Ademáis, os beneficios de UPC en termos de programabilidade e produtividade dos programadores non son facilmente cuantificables. Por outro lado, os compiladores e ambientes de execución para UPC deben tratar con diversos problemas para dar soporte ás funcionalidades PGAS da linguaxe, especialmente o feito da tradución de direccións de memoria compartida a memoria física e, en xeral, o almacenamento de variables para a adecuación a arquitecturas multinúcleo. Nos últimos anos, a implementación a baixo nivel destas funcionalidades mostrou unha notable mellora, e na actualidade diversas implementacións de UPC obteñen un rendemento moi competitivo na transferencia de datos en memoria compartida.

Con todo, o feito de focalizar os esforzos de investigación en UPC na optimización do rendemento da linguaxe causou un estancamento no desenvolvemento da programabilidade, de xeito que a procura dunha maior eficiencia en UPC aínda non se traduciu nunha maior popularidade da linguaxe.

Neste contexto, o primeiro obxectivo da presente Tese de Doutoramento é facer unha análise do "estado da arte" da programación paralela con UPC. A realización dunha análise de rendemento actualizada sobre distintas arquitecturas híbridas de memoria compartida/distribuída, avaliando diversas funcionalidades de UPC, compleméntase co estudo de programabilidade da linguaxe, facendo fincapé no uso de técnicas de análise empírica para obter indicios sobre a percepción da linguaxe a través dos propios programadores. En base a estas tarefas, o seguinte paso nesta Tese é a detección de necesidades inmediatas da linguaxe no eido da produtividade focalizando no ámbito das comunicacións locais e remotas entre *threads*, sendo neste punto confirmadas unha serie de carencias na implementación de funcións colectivas en UPC. Como solución a estes problemas, proxéctase o desenvolvemento dunha biblioteca de funcións colectivas que permita dar solución a unha serie de limitacións nas bibliotecas estándar da linguaxe. Entre as principais aportacións desta biblioteca está a implementación de distintos algoritmos de comunicacións segundo o ambiente de execución, alén dunha interface de usuario flexible para obter unha maior funcionalidade e comodidade para o seu uso. Como punto final, esta Tese presenta o deseño e implementación en paralelo de distintas aplicacións científicas utilizando a devandita biblioteca, facendo unha avaliación do seu rendemento en termos do impacto da introdución das funcións colectivas nos códigos e da escalabilidade obtida para un número elevado de *threads* (até 2048). Para concluír esta Tese, dase unha perspectiva do traballo presente e futuro no ámbito da programación con UPC, tomando como base a experiencia do traballo realizado.

Metodoloxía de Traballo

A metodoloxía utilizada para o desenvolvemento desta Tese de Doutoramento baseouse na definición dunha serie de tarefas, cuns obxectivos iniciais ben determinados. A súa concreción tomou como referencia un estudo do "estado da arte" no ámbito de UPC, tanto en termos de rendemento como de programabilidade, e a

partir deste obxectivo inicial planeouse un obxectivo final para o traballo: o desenvolvemento dunha nova biblioteca de funcións colectivas para a linguaxe UPC que incrementase a produtividade dos programadores. Con todo, os obxectivos intermedios desta Tese definíronse dunha forma ampla, de xeito que fosen as primeiras etapas da investigación as que permitisen unha maior concreción no seu contido en función das necesidades detectadas en tarefas previas, o cal serviría para modelar o contido dos desenvolvementos subsecuentes. A duración destas tarefas estimouse tendo en conta estas posibles variacións de contido, sempre tendo en conta as dependencias entre elas para a súa realización. Finalmente, o traballo realizado distribuiuse en bloques temáticos (Bn), de xeito que cada bloque permitise acadar uns determinados obxectivos (On) mediante a realización dun conxunto de tarefas (Tn). A continuación móstranse a organización das tarefas realizadas seguindo este esquema:

B1 Introdución á linguaxe UPC.

- ***O1.1*** Familiarización co modelo de programación PGAS e a linguaxe UPC.
  - *T1.1.1* Lectura de bibliografía relacionada.
  - *T1.1.2* Aprendizaxe da estrutura dun programa en UPC.
- ***O1.2*** Análise de distintos compiladores e ambientes de execución para UPC.
  - *T1.2.1* Coñecemento dos compiladores de UPC: execución de códigos de proba con distintas implementacións e análise preliminar das súas características.
  - *T1.2.2* Probas de integridade dos códigos: comprobación da corrección nas comunicacións implementadas.
  - *T1.2.3* Probas de comunicacións en memoria compartida (nivel intranodo).
  - *T1.2.4* Execucións sobre diferentes redes de comunicacións en memoria distribuída (nivel internodo).
  - *T1.2.5* Avaliación preliminar dos resultados obtidos: selección dos compiladores e ambientes máis relevantes.

B2 Análise de rendemento e programabilidade da linguaxe UPC.

- ***O2.1*** Avaliación do rendemento das aplicacións sobre diversos sistemas en base ao coñecemento básico adquirido nas tarefas previas.

T2.1.1 Selección de códigos para a análise de rendemento: primitivas de comunicación e códigos de pequeno tamaño que realizan operacións simples (*kernels*).

T2.1.2 Deseño de experimentos: tamaño de problema e número de *threads* usados para as execucións en función dos recursos dispoñibles.

T2.1.3 Execución e depuración dos códigos en ambientes multinúcleo.

T2.1.4 Avaliación de rendemento e estudo de problemas de rendemento e puntos de estrangulamento (*bottlenecks*).

O2.2 Estudo da programabilidade de UPC con base analítica e empírica.

T2.2.1 Análise de estudos previos de programabilidade para linguaxes de programación paralela.

T2.2.2 Deseño e realización dunha proba empírica con programadores nóveis e expertos.

T2.2.3 Obtención e análise dos resultados recollidos: probas de programabilidade e enquisa final.

O2.3 Detección de carencias e necesidades na linguaxe UPC para a realización de futuras accións que proporcionen solucións adecuadas.

T2.3.1 Avaliación dos principais problemas detectados na análise de rendemento: as comunicacións son tomadas como punto de referencia.

T2.3.2 Estudo e análise das carencias da linguaxe UPC reportadas na proba de programabilidade: nomeadamente, o uso de construcións de baixo nivel para obter rendemento.

T2.3.3 Detección de áreas de mellora para a linguaxe UPC: selección da biblioteca de funcións colectivas como principal obxectivo.

B3 Deseño e implementación dunha biblioteca de funcións colectivas estendidas.

O3.1 Organización da biblioteca por grupos de funcións e deseño de interfaces.

T3.1.1 Estudo de traballos previos na área de funcións colectivas en UPC.

T3.1.2 Detección de necesidades específicas para a mellora da biblioteca estándar de colectivas: análise de propostas feitas pola comunidade de usuarios de UPC.

T3.1.3 Estruturación da biblioteca por grupos de funcións.

T3.1.4 Definición das características principais das interfaces: simplicidade e flexibilidade.

O3.2 Desenvolvemento dos algoritmos de funcións colectivas estendidas.

T3.2.1 Desenvolvemento de algoritmos de referencia para cada grupo de funcións: *in-place*, *vector-variant*, *team-based* e *get-put-priv*.

T3.2.2 Optimización dos algoritmos para memoria compartida: uso de versións *get* e *put* para a mellora do rendemento.

T3.2.3 Deseño e implementación de algoritmos para memoria híbrida: aproveitamento das arquitecturas multinúcleo.

T3.2.4 Realización de tests de integridade para as funcións implementadas.

O3.3 Análise de rendemento e programabilidade da biblioteca de funcións implementada.

T3.3.1 Avaliación de rendemento (*microbenchmarking*) das funcións.

T3.3.2 Estudo preliminar do impacto do uso das funcións colectivas estendidas en diferentes *kernels*.

B4 Aplicación da biblioteca de funcións colectivas estendidas en códigos representativos en computación de altas prestacións.

O4.1 Implementación de MapReduce en UPC con operacións colectivas estendidas.

T4.1.1 Análise de implementacións existentes de MapReduce en Java e C.

T4.1.2 Deseño de interfaces e xustificación da utilidade das funcións colectivas estendidas.

T4.1.3 Desenvolvemento do código paralelo: implementación xenérica da funcionalidade con procedementos *template*.

T4.1.4 Análise de rendemento e estudo do impacto das funcións colectivas estendidas.

O4.2 Paralelización do movemento browniano de partículas en UPC con operacións colectivas estendidas.

T4.2.1 Análise bibliográfica e revisión do código de simulación.

T4.2.2 Estudo da distribución de carga de traballo para o código paralelo: análise de dependencias na simulación utilizando distintos algoritmos.

T4.2.3 Implementación do código: uso de distintas reparticióńs de traballo con funcións colectivas estendidas segundo o algoritmo utilizado.

T4.2.4 Avaliación do rendemento obtido polos códigos implementados.

B5 Conclusións tiradas do traballo feito e análise de desenvolvementos futuros.

O5.1 Exposición das principais leccións aprendidas do traballo realizado.

T5.1.1 Resumo do traballo feito e principais aportacións.

T5.1.2 Análise de perspectivas de traballo futuro no eido da linguaxe UPC.

O5.2 Composición da memoria final da Tese de Doutoramento.

T5.2.1 Estruturación e organización da información sobre o traballo feito.

T5.2.2 Redacción da memoria.

Para a elaboración desta Tese de Doutoramento utilizáronse os medios descritos deseguido:

- Material de traballo e financiamento económico fornecidos básicamente polo Grupo de Arquitectura de Computadores da Universidade da Coruña, alén da Xunta de Galicia (contrato predoutoral "María Barbeito") e a Universidade da Coruña (contratos de profesor interino de substitución).

- Proxectos de investigación que teñen financiado esta Tese de Doutoramento:

  - Con financiamento privado: proxecto "Improving UPC Usability and Performance in Constellation Systems: Implementation/Extension of UPC Libraries" (Hewlett-Packard Española, S.L.).

  - Con financiamento estatal (Goberno de España): proxectos TIN2007-67537-C03-02 (Ministerio de Educación e Ciencia) e TIN2010-16735 (Ministerio de Ciencia e Innovación).

- Con financiamento autonómico (Xunta de Galicia): programa de Consolidación de Grupos de Investigación Competitivos (ref. 2006/3 no DOG 13/12/2006 e ref. 2010/06 no DOG 15/09/2010) e Rede Galega de Computación de Altas Prestacións (ref. 2007/147 no DOG 31/08/2007 e ref. 2010/53 no DOG 15/09/2010), alén do devandito contrato predoutoral "María Barbeito" (DOG 15/01/2010, con prórroga no DOG 13/01/2012).

- Clusters e supercomputadores utilizados como usuario:

  - Clúster *muxia* (Universidade da Coruña, 2006-2008): 24 nodos con procesador Intel Xeon (8 nodos a 1.8 GHz con 1 GB RAM, 8 nodos a 2.8 GHz con 2 GB de RAM e 8 nodos a 3.2 GHz con 4 GB de RAM) con redes de interconexión SCI e Gigabit Ethernet.
  - Clúster *nm* (Universidade da Coruña, 2009-2011): 8 nodos con 2 procesadores Intel Xeon 5060 de 2 núcleos (con *Simultaneous Multithreading*) a 3.2 GHz con 8 GB de RAM, 3 nodos con 2 procesadores Intel Xeon E5440 de 4 núcleos a 2.83GHz con 16 GB de RAM, 4 nodos con 2 procesadores Intel Xeon EMT64 de 2 núcleos (con *Simultaneous Multithreading*) a 3.2 GHz con 32 GB de RAM, todos con rede de interconexión InfiniBand.
  - Clúster *pluton* (Universidade da Coruña, desde 2008): 8 nodos con 2 procesadores Intel Xeon E5620 de 4 núcleos a 2.4 GHz con até 16 GB de RAM, 8 nodos con 2 procesadores Intel Xeon E5520 de 4 núcleos a 2.27 GHz con até 8 GB de RAM e 2 nodos con 2 procesadores Intel Xeon E5440 de 4 núcleos a 2.83 GHz con até 8 GB de RAM, todos con rede de interconexión InfiniBand. Ademáis, 1 nodo con 4 procesadores Intel Xeon E7450 de 6 núcleos a 2.4 GHz e 32 GB de RAM.
  - Supercomputador *Finis Terrae* (Centro de Supercomputación de Galicia, desde 2008): 144 nodos con procesador Itanium2 Montvale de 16 núcleos a 1.6 GHz con 128 GB de RAM e rede de interconexión InfiniBand 4 x DDR a 20 Gbps, e 1 nodo Superdome con procesador Itanium2 Montvale de 128 núcleos a 1.6 GHz con 1 TB de RAM. Estaba clasificado no posto 101 da lista TOP500 no momento de comezar a súa utilización, segundo os datos de Novembro de 2007 (na actualidade non está incluído na devandita lista).

  - Supercomputador *SVG* (Centro de Supercomputación de Galicia, desde 2011): 46 nodos HP ProLiant SL165z G7 con 2 procesadores AMD Opteron 6174 de 12 núcleos, 2.2 GHz e até 64 GB de RAM, e rede de interconexión Gigabit Ethernet.
  - Supercomputador *JuRoPa* (Forschungszentrum Jülich, desde 2011): 2208 nodos con 2 procesadores Intel Xeon X5570 (Nehalem-EP) de 4 núcleos a 2.93 GHz con 24 GB de RAM e rede de interconexión InfiniBand. Estaba clasificado no posto 25 da lista TOP500 no momento de comezar a súa utilización, segundo os datos de Xuño de 2011 (na actualidade ocupa o posto 89 da mesma lista, conforme á actualización de Novembro de 2012).

- Estadía de investigación no Jülich Supercomputing Centre (Forschungszentrum Jülich, Alemaña) durante 3 meses (do 1 de setembro ao 30 de novembro de 2011), a cal permitiu a realización das tarefas asociadas ao obxectivo 4.2 en colaboración co Profesor Godehard Sutmann. A realización desta colaboración facilitou o acceso ao supercomputador JuRoPa referido anteriormente. O financiamento desta estadía facilitouno a Xunta de Galicia na súa convocatoria de axudas para estadías para contratados predoutorais "María Barbeito" (DOG 21/12/2011).

Conclusións

A presente Tese de Doutoramento, *"Extended Collectives Library for Unified Parallel C"*, analizou os actuais desenvolvementos e características proporcionados pola linguaxe UPC. O modelo de memoria PGAS mostrou ser beneficioso para a programabilidade de aplicacións para computación de altas prestacións, e consecuentemente o centro da atención da Tese púxose na avaliación da utilidade das propiedades PGAS en UPC e no desenvolvemento de novas funcionalidades para favorecer a produtividade.

A primeira etapa no desenvolvemento da tese consistiu na análise da linguaxe UPC en termos de rendemento e programabilidade. A principal fonte de información nesas áreas foi a avaliación de rendemento de funcións UPC mediante *microbenchmarking* e a análise de programabilidade mediante probas empíricas realizadas con

dous grupos heteroxéneos de programadores. O estudo dos códigos desenvolvidos polos participantes nas sesións e as súas valoracións sobre a experiencia con UPC, xunto cunha avaliación xeral das funcionalidades de UPC, proporcionaron información moi relevante para definir varias áreas de mellora:

- A obtención dun bo rendemento tende a estar en conflito co uso de construcións para programabilidade: varios elementos da linguaxe, como as asignacións a variables compartidas, non obteñen un rendemento óptimo e por iso substitúense por solucións máis eficientes que introducen maior complexidade no código, como as privatizacións de variables. A solución máis conveniente sería unha implementación eficiente integrada no compilador, pero o uso de movementos de datos pequenos é en xeral difícil de optimizar.
- Algunhas das funcións nas bibliotecas estándar, como as funcións colectivas, presentan múltiples restricións na súa aplicabilidade a códigos paralelos. Por exemplo, as colectivas estándar só poden definir comunicacións entre variables compartidas, e o tamaño dos datos enviados por cada *thread* debe ser igual. Outro efecto colateral é que as variables privatizadas non poden ser usadas en comunicacións colectivas, o cal tamén inflúe no rendemento.
- Hai unha carencia de desenvolvementos para obter maior programabilidade: a mellora das implementacións da linguaxe a baixo nivel foi o principal obxectivo das investigacións no eido de UPC nos últimos anos, e a programabilidade asumiuse como un factor implícito. Aínda que isto sexa certo en comparación con outras aproximacións (por exemplo, MPI), é preciso explorar a posibilidade de ofrecer bibliotecas adicionais ou funcionalidades de alto nivel para simplificar a escritura de código.

De acordo con estes feitos, esta Tese procurou dar solución a estes problemas en termos de programabilidade e sen esquecer o rendemento. Por iso, a segunda etapa no desenvolvemento da Tese foi a mellora da programabilidade en UPC por medio dunha biblioteca estendida de funcións colectivas. Esta biblioteca soluciona as limitacións máis relevantes da especificación de UPC coas seguintes características:

- O uso da mesma variable como orixe e destino das comunicacións (colectivas *in-place*).

- O soporte de distintos tamaños de mensaxe en cada *thread* (colectivas *vector-variant*).
- A selección dun subconxunto dos *threads* nun programa para a execución dunha tarefa específica (colectivas *team-based*).
- O uso de variables privadas como orixe e/ou destino das comunicacións (colectivas *get-put-priv*).

En total, a biblioteca ten cerca de 200 funcións colectivas estendidas que implementan as características anteriores, incluíndo algunhas variacións e combinacións destas e outras funcionalidades, co obxectivo de proporcionar a máxima flexibilidade para as comunicacións colectivas con UPC. Internamente, os algoritmos destas funcións están implementados de maneira eficiente, incluíndo diferentes características adicionais:

- Implementación adaptada á execución en arquitecturas multinúcleo: a dispoñibilidade de varios algoritmos de comunicación colectiva permite a selección do óptimo para o uso en diferentes plataformas. As comunicacións en memoria compartida aprovéitanse a través de transferencias de datos baseadas en árbores planas, mentres que as comunicacións entre nodos utilizan árbores binomiais para aproveitar a largura de banda máis adecuadamente. As aproximacións híbridas utilízanse preferentemente en ambientes multinúcleo, xunto con optimizacións no uso da memoria e a asignación de *threads* a núcleos (*thread pinning*) cando for posible.
- Privatización interna de argumentos de colectivas estendidas: a definición de variables orixe e destino, como tamén outros argumentos auxiliares, é maioritariamente feita no espazo de memoria compartido, pero internamente estas variables son privatizadas de xeito transparente ao usuario para obter mellor rendemento.
- Desenvolvemento dunha biblioteca de soporte para grupos de *threads* (*teams*): as colectivas *team-based* constrúense mediante a manipulación dun grupo de *threads* no programa (*team*), pero na actualidade non hai unha definición estándar para esta estrutura. Por iso, desenvolveuse unha biblioteca auxiliar

para soportar a funcionalidade precisa para a implementación das colectivas *team-based*, incluíndo funcións para a creación e destrución de *teams*, ou tamén a inclusión e eliminación de *threads* neles.

O uso da biblioteca de funcións colectivas foi inicialmente testada con catro *kernels* representativos, que realizan operacións comúns como a multiplicación de matrices (densas e dispersas), a ordenación de enteiros ou a transformada de Fourier (FFT) en 3D. A flexibilidade da biblioteca favoreceu a introdución de colectivas nestes códigos, substituíndo as copias explícitas de datos en memoria e a privatización de variables, o cal proporcionou códigos máis simples. Adicionalmente, os algoritmos e optimizacións para ambientes multinúcleo das funcións da biblioteca melloraron o rendemento dos códigos, que en determinados casos mostraron maior eficiencia que códigos análogos implementados con MPI, como no caso da FFT.

A terceira etapa no desenvolvemento da Tese foi a implementación dunha nova funcionalidade para mellorar a programabilidade de UPC, que consistiu no desenvolvemento do *framework* UPC MapReduce (UPC-MR). UPC-MR representa a adaptación para UPC do coñecido paradigma de procesado masivo de datos de baixa granularidade, e foi concebido para obter programabilidade asumindo tres principios básicos:

- O usuario non precisa escribir código paralelo: UPC-MR proporciona dúas funcións modelo (*template*) para a xestión das etapas "Map" e "Reduce", respectivamente, para facilitar a interface e dando a posibilidade de realizar todas as comunicacións entre *threads* de forma transparente ao usuario.

- A implementación é xenérica: todos os elementos de entrada ás dúas funcións de xestión son procesados de xeito análogo, independentemente do seu tipo de datos. O único requerimento é que as funcións definidas polo usuario para "Map" e "Reduce" deben ter un tipo de datos coherente para un correcto procesado dos valores de entrada e saída.

- O procesamento optimízase de acordo cos requerimentos de comunicacións: UPC-MR proporciona a flexibilidade suficiente para realizar axustes concretos na especificación das comunicacións. O uso de diferentes opcións de configura-

ción, así como da biblioteca de funcións colectivas, axuda a mellorar a xestión dos datos e das comunicacións, respectivamente.

Finalmente, a última etapa da elaboración da Tese consistiu na análise e avaliación de aplicacións de grande tamaño que aproveitan os desenvolvementos anteriores (a biblioteca estendida de colectivas UPC e o *framework* UPC-MR) para grandes sistemas computacionais, co obxectivo de obter escalabilidade para un grande número de *threads* (até 2048). Esta avaliación incluiu: (1) varios códigos con procesamento de tipo MapReduce, e (2) a paralelización dunha simulación de movemento browniano de partículas (*Brownian dynamics*).

As aplicacións utilizadas para testar UPC-MR presentaron diferentes cargas computacionais para as etapas "Map" e "Reduce", así como un número de elementos de entrada diferentes, co obxectivo de analizar o impacto destas características no rendemento dos distintos códigos e para despois comparalos con outras implementacións de MapReduce en memoria compartida e distribuída utilizando C (Phoenix) e MPI (MapReduce-MPI), respectivamente. As principais conclusións desta avaliación de rendemento son as seguintes:

- A cantidade de procesamento asociada a cada elemento de entrada na etapa "Map" é un factor clave para o rendemento xeral: se é moi reducida, o procesado un a un de cada elemento (*element-by-element*) na implementación de MPI resulta ser máis ineficiente que o de UPC en memoria distribuída, especialmente cando o número de elementos de entrada é elevado. O código de memoria compartida implementado con Phoenix obtén mellores resultados, aínda que o uso repetido de chamadas a funcións de bibliotecas en C++ provoca menor rendemento que o código UPC.
- O uso das colectivas estendidas pode axudar a optimizar o rendemento, especialmente cando se utiliza un número elevado de *threads* e consecuentemente hai unha grande cantidade de comunicacións na etapa "Reduce". Nestes casos de proba mostrouse a posibilidade de obter escalabilidade até millares de *threads* dependendo do tempo consumido por "Reduce" en relación co tempo da etapa "Map".

En relación á simulación do movemento browniano de partículas, desenvolvéronse

tres implementacións usando OpenMP, MPI e UPC para comparar o seu rendemento en diferentes situacións. En xeral, a implementación en OpenMP ofrece a maior programabilidade mediante o uso de directivas, aínda que o seu rendemento limítase a execucións intranodo sen ter un aproveitamento axeitado para sistemas de memoria distribuída, de xeito análogo á paralelización directa sobre memoria compartida usando UPC. O uso dunha distribución de datos axeitada proporciona un alto rendemento e escalabilidade para MPI e UPC. Os principais resultados extraídos deste traballo son os seguintes:

- A análise do código secuencial e das súas dependencias, que axudaron a detectar as principais partes do algoritmo de simulación e guiaron o proceso de descomposición do dominio para cada ambiente de execución.
- O uso das colectivas estendidas, que melloraron o rendemento de UPC e favoreceron a súa escalabilidade até millares de *threads*.
- Unha única implementación coa linguaxe UPC foi capaz de obter alto rendemento tanto nun ambiente de memoria compartida como de memoria distribuída, con resultados similares ou incluso mellores que os dos códigos de OpenMP e MPI, respectivamente para cada ambiente de execución.

As contribucións desta Tese publicáronse en varias revistas e congresos con revisión por pares. As avaliacións iniciais de UPC, tanto de rendemento [44, 46, 77] como de programabilidade [83], levaron ao deseño e implementación da biblioteca de funcións colectivas estendidas [82], que foron aplicadas con éxito a unha implementación eficiente do *framework* MapReduce con UPC [81] e á paralelización da simulación do movemento browniano de partículas [78, 79, 80].

Entre as principais liñas de investigación para estender este traballo, cabe mencionar a integración das funcionalidades propostas nos compiladores de UPC existentes na actualidade, como un primeiro paso para a súa aceptación e espallamento na comunidade de programadores e usuarios de UPC. A vindeira especificación estándar da linguaxe UPC definirá un novo marco para os futuros desenvolvementos na linguaxe, e neste punto as colectivas estendidas poden usarse como unha implementación de referencia que mostra os beneficios e posibilidades da estensión do ámbito das colectivas. Deste xeito, e cos códigos desenvolvidos e avaliados nesta

Tese, poderase debater a integración da biblioteca estendida (ou dun subconxunto dela) como unha biblioteca requerida ou opcional en futuras especificacións de UPC.

Principais Contribucións

En termos xerais, as contribucións máis relevantes deste traballo son as seguintes:

- A descripción detallada da linguaxe UPC como alternativa a solucións de programación paralela máis tradicionais (MPI e OpenMP) para o desenvolvemento de códigos en arquitecturas de memoria híbrida compartida/distribuída. A análise das súas vantaxes e inconvenientes en comparación con outras aproximacións, con especial atención ao estudo do modelo de memoria PGAS, xustifica a utilización de UPC pola súa capacidade de adaptación aos devanditos ambientes.

- A análise actualizada do rendemento da linguaxe UPC, a través do estudo de primitivas básicas (funcións de movemento de datos e funcións colectivas) e distintos *kernels* e aplicacións. O uso de diferentes compiladores, alén de varios ambientes clúster e supercomputadores, axuda a ofrecer unha maior variedade de resultados e aumenta a significatividade nas avaliacións.

- O estudo da linguaxe UPC en termos de programabilidade e produtividade mediante a análise empírica con programadores expertos e inexpertos en programación paralela. O desenvolvemento de códigos nun ambiente de probas adecuadamente illado, complementado coa información reportada polos participantes na proba, favoreceu a obtención de valiosa información sobre a potencialidade de UPC e as áreas de mellora máis importantes.

- O desenvolvemento dunha biblioteca de funcións colectivas estendidas, ofrecendo unha sintaxe simple e flexible para a implementación de movementos de datos en UPC de xeito máis produtivo. Estas funcións desenvolvéronse utilizando elementos de sintaxe estándar de UPC e ANSI C, proporcionando distintos algoritmos configurables (de forma automática ou a nivel de interface) que permiten obter un rendemento optimizado en ambientes de memoria

compartida e memoria híbrida compartida/distribuída. Ademáis, en certos casos de operacións de tipo *in-place*, o seu rendemento móstrase notablemente mellor en memoria distribuída que o de implementacións paralelas tradicionais como MPI, alén de proporcionar unha maior escalabilidade de xeito que a melloría incrementa co número de *threads* e o tamaño da mensaxe.

- A implementación do modelo de programación MapReduce para o procesamento masivo de datos en paralelo con UPC. A abstracción das comunicacións e a súa capacidade de procesamento xenérico de datos posibilita a súa integración en calquer tipo de aplicación existente, tanto secuencial en C como paralela en UPC. Alén diso, a utilización das novas funcións colectivas estendidas axuda a obter códigos máis eficientes en ambientes de computación de altas prestacións.

- A avaliación das novas funcionalidades mediante a súa utilización en códigos científicos e computacionais. O principal desenvolvemento neste ámbito é a paralelización da simulación do movemento browniano de partículas: a análise de dependencias, a adaptación da estrutura da aplicación a ambientes de execución paralela e a utilización de funcións colectivas estendidas contribuíron a obter un desenvolvemento eficiente, mostrando un rendemento competitivo con outras aproximacións análogas usando MPI e OpenMP, tamén desenvolvidas no ámbito desta Tese de Doutoramento.

Publications from the Thesis

Journal Papers (3)

- Carlos Teijeiro, Godehard Sutmann, Guillermo L. Taboada, Juan Touriño. Parallel Brownian Dynamics Simulations with the Message-Passing and PGAS Programming Models. *Computer Physics Communications*, 2013 (In press) (**Journal in JCR**).
  DOI: 10.1016/j.cpc.2012.12.015

- Carlos Teijeiro, Godehard Sutmann, Guillermo L. Taboada, Juan Touriño. Parallel Simulation of Brownian Dynamics on Shared Memory Systems with OpenMP and Unified Parallel C. *Journal of Supercomputing*, 2013 (In press) (**Journal in JCR**).
  DOI: 10.1007/s11227-012-0843-1

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Ramón Doallo, José C. Mouriño, Damián A. Mallón, Brian Wibecan. Design and Implementation of an Extended Collectives Library for Unified Parallel C. *Journal of Computer Science and Technology*, 28(1):72–89, 2013 (**Journal in JCR**).
  DOI: 10.1007/s11390-013-1313-9

International Conferences - full papers (6)

- Carlos Teijeiro, Godehard Sutmann, Guillermo L. Taboada, Juan Touriño. Parallelization and Performance Analysis of a Brownian Dynamics Simulation using OpenMP. In *Proc. 12th International Conference on Computational and*

Mathematical Methods in Science and Engineering (CMMSE'12), pp. 1143–1154, La Manga del Mar Menor (Murcia), Spain, 2012.
ISBN: 978-84-615-5392-1

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Ramón Doallo. Design and Implementation of MapReduce using the PGAS Programming Model with UPC. In *Proc. 17th International Conference on Parallel and Distributed Systems (ICPADS'11)*, pp. 196–203, Tainan, Taiwan, 2011.
DOI: 10.1109/ICPADS.2011.162

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, Damián A. Mallón, Andrés Gómez, José C. Mouriño, Brian Wibecan. Evaluation of UPC Programmability using Classroom Studies. In *Proc. 3rd Conference on Partitioned Global Address Space Programming Models (PGAS'09)*, pp. 10:1–10:7, Ashburn (VA), USA, 2009.
DOI: 10.1145/1809961.1809975

- Damián A. Mallón, Andrés Gómez, José C. Mouriño, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, Brian Wibecan. UPC Performance Evaluation on a Multicore System. In *Proc. 3rd Conference on Partitioned Global Address Space Programming Models (PGAS'09)*, pp. 9:1–9:7, Ashburn (VA), USA, 2009.
DOI: 10.1145/1809961.1809974

- Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, José C. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Lecture Notes in Computer Science vol. 5759*, pp. 174–184, Espoo, Finland, 2009.
DOI: 10.1007/978-3-642-03770-2_24

- Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, José C. Mouriño, Damián A. Mallón, Andrés Gómez. Performance Evaluation of Unified Parallel C Collective Communications. In *Proc. 11th IEEE International Conference on High Performance Computing*

and Communications (HPCC'09), pp. 69–78, Seoul, Korea, 2009.
DOI: 10.1109/HPCC.2009.88

International Conferences - posters (1)

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

National Conferences - full papers (2)

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Ramón Doallo. Diseño e Implementación de MapReduce en el Modelo de Programación PGAS. In *Actas de las XXI Jornadas de Paralelismo (JP'10) - CEDI 2010*, pp. 643–650, Valencia, Spain, 2010.
ISBN: 978-84-92812-49-3

- Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, Jose C. Mouriño, Andrés Gómez, Ignacio L. Cabido. Evaluación del Rendimiento de las Comunicaciones Colectivas en UPC. In *Actas de las XIX Jornadas de Paralelismo (JP'08)*, pp. 399–404, Castellón de la Plana, Spain, 2008.
ISBN: 978-84-8021-676-0

Abstract

Current multicore processors mitigate single-core processor problems (e.g., power, memory and instruction-level parallelism walls), but they have raised the programmability wall. In this scenario, the use of a suitable parallel programming model is key to facilitate a paradigm shift from sequential application development while maximizing the productivity of code developers. At this point, the PGAS (Partitioned Global Address Space) paradigm represents a relevant research advance for its application to multicore systems, as its memory model, with a shared memory view while providing private memory for taking advantage of data locality, mimics the memory structure provided by these architectures. Unified Parallel C (UPC), a PGAS-based extension of ANSI C, has been grabbing the attention of developers for the last years. Nevertheless, the focus on improving performance of current UPC compilers/runtimes has been relegating the goal of providing higher programmability, where the available constructs have not always guaranteed good performance.

Therefore, this Thesis focuses on making original contributions to the state of the art of UPC programmability by means of two main tasks: (1) presenting an analytical and empirical study of the features of the language, and (2) providing new functionalities that favor programmability, while not hampering performance. Thus, the main contribution of this Thesis is the development of a library of extended collective functions, which complements and improves the existing UPC standard library with programmable constructs based on efficient algorithms. A UPC MapReduce framework (UPC-MR) has also been implemented to support this highly scalable computing model for UPC applications. Finally, the analysis and development of relevant kernels and applications (e.g., a large parallel particle simulation based on Brownian dynamics) confirm the usability of these libraries, concluding that UPC can provide high performance and scalability, especially for environments with a large number of threads at a competitive development cost.

Para quem estava

Para quen está

Para quem estiver

Acknowledgments

The development of this Thesis has been the result of a gratifying personal experience, which has been possible thanks not only to my own work, but also to the support of many people that have been with me in this period of my life. I would like to acknowledge my advisors Guillermo and Juan for their guidance and patience during this time. Big thanks also go to all the colleagues and comrades-in-arms at the Computer Architecture Group, extending this to all the inhabitants of Lab. 1.2 that have seen me there in these years, for all the good times.

I also want to acknowledge the people at the Supercomputing Center of Galicia (CESGA), the University of Santiago de Compostela (USC) and Hewlett-Packard (HP) that I met during my doctoral courses and with whom I worked in different projects and tasks. A particular acknowledgment goes to the systems support staff at CESGA, for rebooting the nodes of the Finis Terrae supercomputer after the crash of many of my submitted jobs.

Very special thanks go to Godehard Sutmann for hosting me during my research visit to the Jülich Supercomputing Centre (JSC) at Forschungszentrum Jülich, making this extensive to all the people that have been with me during those very inspiring months in Germany (and more).

Additionally, I would like to thank the institutions and projects that have funded my work on research and teaching during these years. I specially thank the Computer Architecture Group and the Department of Electronics and Systems at the University of A Coruña for providing the necessary tools for my everyday work, as well as CESGA and JSC for providing access to the supercomputers that have been used as testbeds for this work.

Carlos Teijeiro Barjas

...

...

Enjoy the silence

...

...

Contents

Preface **1**

1. Parallel Programming with Unified Parallel C **9**

1.1. Introduction to UPC 9

1.2. UPC Language and Specification 11

1.3. UPC Compiler/Runtime Projects 13

1.3.1. Open Source Projects 13

1.3.2. Vendor Projects 14

1.4. Comments on UPC Programmability 15

2. UPC Productivity Analysis **17**

2.1. Programmability vs. Performance with UPC 18

2.1.1. Privatization of Shared Variables 19

2.1.2. UPC Standard Collective Operations 21

2.2. Empirical Programmability Analysis of UPC with Classroom Studies 25

2.2.1. Design of the Programmability Study 27

2.2.2. Analysis of Results 29

2.2.3. Reported Comments on UPC 35

2.3. Proposal of Programmability Improvements 38

3. Design and Implementation of Extended Collectives in UPC 41

3.1. Design of the Collectives Library 42

3.2. Implementation of Extended Collectives 44

3.2.1. In-place Collectives 44

3.2.2. Vector-variant Collectives 48

3.2.3. Team-based Collectives 53

3.2.4. Get-put-priv Collectives 58

3.3. Optimization of Extended Collectives 60

3.4. Use of Extended Collectives: Case Studies 65

3.4.1. Dense Matrix Multiplication Kernel 65

3.4.2. Sparse Matrix Multiplication Kernel 68

3.4.3. Integer Sort Kernel 69

3.4.4. 3D Fast Fourier Transform Kernel 70

3.5. Microbenchmarking of UPC Collectives 72

3.6. Performance Evaluation of UPC Kernels 77

3.7. Conclusions to Chapter 3 84

4. MapReduce for UPC: UPC-MR 87

4.1. Introduction to MapReduce 88

4.2. Design and Implementation of UPC-MR 88

4.2.1. Function `ApplyMap` 90

4.2.2. Function `ApplyReduce` 92

4.3. Performance Evaluation 94

4.3.1. Experimental Conditions 94
4.3.2. Performance Results 96
4.4. Conclusions to Chapter 4 103

5. Parallel Simulation of Brownian Dynamics 105
5.1. Introduction to the Simulation of Brownian Dynamics 106
5.2. Theoretical Basis of Brownian Dynamics 107
5.3. Implementation of the Simulation Code 109
5.4. Development of the Parallel Implementation 113
5.4.1. Parallel Workload Decomposition 113
5.4.2. Shared Memory Parallelization (UPC & OpenMP) 114
5.4.3. Distributed Memory Parallelization (UPC & MPI) 118
5.4.4. Optimizations for Distributed Memory 119
5.5. Performance Evaluation 125
5.6. Conclusions to Chapter 5 135

Conclusions 137

A. API of the UPC Extended Collectives Library 143
A.1. In-place Collectives 144
A.1.1. The `upc_all_broadcast_in_place` Collective 144
A.1.2. The `upc_all_scatter_in_place` Collective 145
A.1.3. The `upc_all_gather_in_place` Collective 147
A.1.4. The `upc_all_gather_all_in_place` Collective 149
A.1.5. The `upc_all_exchange_in_place` Collective 150

A.1.6. The upc_all_permute_in_place Collective 152
A.1.7. Computational In-place Collectives 154
A.2. Vector-variant Collectives 158
A.2.1. The upc_all_broadcast_v Collective 158
A.2.2. The upc_all_scatter_v Collective 163
A.2.3. The upc_all_gather_v Collective 168
A.2.4. The upc_all_gather_all_v Collective 173
A.2.5. The upc_all_exchange_v Collective 178
A.2.6. The upc_all_permute_v Collective 185
A.2.7. The upc_all_vector_copy Collective 189
A.2.8. Computational Vector-variant Collectives 194
A.3. Team-based Collectives 200
A.3.1. The upc_all_broadcast_team Collective 200
A.3.2. The upc_all_scatter_team Collective 202
A.3.3. The upc_all_gather_team Collective 204
A.3.4. The upc_all_gather_all_team Collective 207
A.3.5. The upc_all_exchange_team Collective 209
A.3.6. The upc_all_permute_team Collective 211
A.3.7. Computational Team-based Collectives 214
A.4. Get-put-priv Variants of Standard Collectives 219

References **223**

List of Tables

2.1. UPC vs. MPI collectives performance (32 threads/processes, MPI = 100%) 24

2.2. Summary of codes obtained in the classroom studies 29

2.3. Stencil with 10^6 elements (UDC group) 30

2.4. Buffon-Laplace Needle with 10^7 trials (UDC group) 32

2.5. Minimum Distance Computation with 500 nodes (UDC group) 32

2.6. Stencil with 10^6 elements (CESGA group) 33

2.7. Buffon-Laplace Needle with 10^7 trials (CESGA group) 33

2.8. Minimum Distance Computation with 500 nodes (CESGA group) . . 34

2.9. Minimum Distance Computation - MPI vs. UPC 35

5.1. Breakdown of the execution time of the sequential code 112

List of Figures

1.1. Scheme of the PGAS memory model in UPC 12

2.1. Collective communications: flat tree vs. binomial tree (8 threads) . . 23

2.2. Reported benefits and drawbacks of UPC 36

2.3. Most urgent reported issue to solve 37

3.1. Scheme of the Extended Collectives library 43

3.2. Communications for `upc_all_gather_all_in_place` (3 threads) . . . 46

3.3. Communications for `upc_all_exchange_in_place` (4 threads) 46

3.4. Communications for `upc_all_reduceI_v` (4 threads) 50

3.5. Communications for different versions of `upc_all_broadcast_v` (4 threads) 52

3.6. Communications for team-based scatter operations (4 threads) 56

3.7. Data movements for `upc_all_broadcast_priv` (4 threads) 59

3.8. Data movements for `upc_all_scatter_priv` with flat-tree algorithm on 4 nodes 62

3.9. Data movements for `upc_all_scatter_priv` with binomial-tree algorithm on 4 nodes 63

3.10. Data movements for `upc_all_scatter_priv` with binomial-flat-tree algorithm on 4 nodes 64

3.11. Communications for `upc_all_exchange_v_merge_local` (2 threads) . 71

3.12. Performance of `upc_all_broadcast_in_place` 75

3.13. Performance of `upc_all_scatter_v` 76

3.14. Performance of `upc_all_exchange_in_place` 76

3.15. Performance of dense matrix multiplication (4480×4480) 78

3.16. Performance of sparse matrix multiplication (16614×16614 sparse matrix and 16614×4480 dense matrix) 80

3.17. Performance of NPB Integer Sort 81

3.18. Performance of NPB 3D FFT on JRP with 1 thread/node 82

3.19. Performance of NPB 3D FFT on JRP with 8 threads/node 83

4.1. Examples of *BLOCK*, *CYCLIC* and *BALANCED* work distributions 92

4.2. Performance comparison of MapReduce-MPI and UPC-MR on distributed memory 97

4.3. Performance of *Hist* on shared and distributed memory 99

4.4. Performance of *LinReg* on shared and distributed memory 100

4.5. Performance of *SpamC* and *LinReg* on hybrid shared/distributed memory (JRP) 102

5.1. Example of the short range interaction model with periodic boundary conditions 108

5.2. Work and data distribution with `D` as a shared matrix in UPC 115

5.3. Balanced workload decomposition of `D` on distributed memory 118

5.4. Balanced distribution of matrix `D` for Fixman's algorithm (detailed data structures for process 0) 123

5.5. Work distribution with `D` as a full-size private matrix 125

5.6. Shared memory performance results with 256 particles (Fixman and Cholesky) 127

5.7. Shared memory performance results with 4096 particles (Fixman) . . 129

5.8. Distributed memory performance results with 256 particles on JRP . 130

5.9. Distributed memory performance results with 1024 particles on JRP . 132

5.10. Distributed memory performance results with 4096 particles on JRP . 133

5.11. Efficiency comparison between Fixman *bal-comms* and *min-comms* algorithms, expressed as the percentage of the efficiency of the *min-comms* code 134

List of Listings

2.1. Example of programmability vs. performance in UPC 20

2.2. Signatures of the UPC collectives included in the current specification 22

3.1. Signatures of representative in-place collectives 44

3.2. Signature of a representative rooted in-place collective 47

3.3. Time measuring routine using in-place collectives 47

3.4. Signatures of representative vector-variant collectives 48

3.5. Activation of optimized processing for `upc_all_reduceI_v` 50

3.6. Signatures of representative versions of vector-variant collectives . . . 51

3.7. Copy of a triangular matrix using vector-variant collectives 53

3.8. Structure for UPC teams support 54

3.9. Signatures of the team management auxiliary functions 55

3.10. Signature of a team-based collective 56

3.11. Computation of Pi using team-based collectives 57

3.12. Signatures of representative get-put-priv collectives 59

3.13. Image filtering algorithm using get-put-priv collectives 60

3.14. Original UPC dense matrix multiplication code 66

3.15. UPC dense matrix multiplication code with extended collectives . . . 67

3.16. UPC sparse matrix multiplication code with extended collectives . . . 68

3.17. Original UPC code in Integer Sort 70

3.18. UPC code in Integer Sort using extended collectives 70

3.19. Original UPC code in 3D FFT 71

3.20. UPC code in 3D FFT using extended collectives 72

4.1. Signatures of the basic functions in UPC-MR 90

5.1. Iterative Cholesky decomposition with OpenMP 116

5.2. OpenMP pseudocode that computes the maximum eigenvalue of `D` in Fixman's algorithm 117

5.3. Pseudocode for the computation of displacements with Cholesky decomposition (distributed memory) 121

5.4. Pseudocode for the parallel computation of the maximum eigenvalue with Fixman's algorithm (distributed memory) 124

List of Abbreviations

API *Application Programming Interface*

BCOL *Special acronym for the Berkeley UPC COLlectives implementation*

BUPC *Special acronym for the Berkeley UPC compiler and runtime*

CESGA *CEntro de Supercomputación de GAlicia (Galicia Supercomputing Center)*

CSR *Compressed Sparse Row*

DDR *Double Data Rate*

DMP *Special acronym for "Distributed Memory communications"*

FFT *Fast Fourier Transform*

FT *Special acronym for the Finis Terrae supercomputer at CESGA*

GASNet *Global-Address Space Networking*

HCA *Host Channel Adapter*

HPC *High Performance Computing*

HYB *Special acronym for "HYBrid shared/distributed memory communications"*

IA-64 *Intel Itanium microprocessor Architecture (64 bits)*

IBV *Special acronym for the InfiniBand Verbs conduit used by the runtime of BUPC*

IS *Integer Sort*

JRP *Special acronym for the JuRoPa supercomputer at JSC*

JSC *Jülich Supercomputing Centre, at Forschungszentrum Jülich*

MGC *Special acronym for one of the MaGny-Cours nodes of the SVG*

MKL *Intel Math Kernel Library*

MPI *Message-Passing Interface*

MTU *Michigan Technological University*

NAS *NASA Advanced Supercomputing*

NPB *NAS Parallel Benchmark(s)*

OpenMP *Open Multiprocessing*

PGAS *Partitioned Global Address Space*

REF *Special acronym for the REFerence implementation of UPC collectives developed at MTU*

SLOC *Source Lines Of Code*

SMP *Special acronym for "Shared Memory communications" in Chapter 2, and for a cluster node at UDC in Chapter 4*

SMT *Simultaneous Multithreading*

SPMD *Single Program, Multiple Data*

SVG *Special acronym for the SVG supercomputer at CESGA*

UC Berkeley *University of California, Berkeley*

UDC *Universidade Da Coruña (University of A Coruña)*

UPC *Unified Parallel C*

UPC-MR *UPC MapReduce (framework)*

Preface

Currently, the popularization and further development of new hybrid shared/distributed memory architectures based on multicore processors is providing a significantly higher processing power. In this scenario the development of High Performance Computing (HPC) parallel applications is essential in order to take full advantage of the possibilities offered by these architectures. However, the growing complexity of these systems is increasing the difficulties for the development of efficient codes, which demand additional efforts and expertise from the application developers. At this point, traditional parallel languages and libraries, either for distributed memory (e.g., MPI) or shared memory (e.g., OpenMP), are not suitable for providing an integral solution for these environments.

In order to solve this situation, there is a growing interest in the development of new programming paradigms that favor a higher productivity for parallel applications. Among different approaches, the PGAS (Partitioned Global Address Space) paradigm has gained significant popularity, because of its memory view divided in two spaces: shared (visible and accessible for all threads in a program) and private (local-access blocks only accessible by its associated thread). Among the languages that implement the PGAS paradigm, Unified Parallel C (UPC), the parallel extension of the ANSI C language, deserves to be mentioned. UPC includes constructs for parallel programming, being the most relevant: (1) remote data movements by means of assignments to shared variables, (2) predefined constants that determine different parameters of the parallel execution, such as the number of threads; and (3) libraries of functions for data copies and collective operations.

However, despite the suitability of the use of UPC on HPC applications, the language has not been widely adopted, either because of the lack of efficient imple-

mentations of compilers and runtimes or because of some limitations in the expressiveness of some operations. The analysis and improvement of UPC performance has been an outstanding area of research in the last years, and a shared outcome by the research in this topic is the incompatibility of programmable constructs with the achievement of efficient codes. For instance, the use of privatizations for shared arrays is a key factor to avoid the overhead of address translations when accessing shared memory positions. As a result, there is a need for further analyses regarding programmability and performance in order to integrate the efficient UPC implementations into higher-level constructs that provide higher coding expressiveness and hence parallel programming productivity. The controversy between the efficiency and the simplicity of UPC codes also indicates that the UPC specification (whose current version dates from 2005) still needs some further improvements in order to enhance the flexibility of the existing UPC constructs and libraries. Thus, the productivity of UPC developments will be significantly increased by providing higher level functionalities based on efficient algorithms that take advantage of data locality in multicore architectures.

Work Methodology

Currently the UPC community is involved in the process of updating the UPC specification. In this context, there is a consensus about the explicit differentiation between language specifications and library specifications, where a library can be considered as required or optional. Given this scenario, this Thesis is intended to contribute to the discussions within the UPC community, endorsing the latest agreements of the community of UPC users and programmers.

The Thesis methodology has followed a classical approach in research and engineering: analysis, design, implementation and evaluation. Thus, the work of this Thesis has started with the study of the state of the art in parallel programming with UPC, evaluating multiple implementations and compilers for the language. The baseline analysis of performance of UPC constructs is here complemented with the evaluation of UPC programmability using analytical and empirical methods, studying the productivity and the impressions of programmers about the UPC language through the analysis of the codes developed by them in a controlled environment

(a type of test known as "classroom study"), as well as a survey of programmers' opinions on UPC benefits and drawbacks. The feedback obtained from these initial tasks has been used to define and structure the subsequent developments, where one of the most relevant contributions is the extension of the UPC standard collectives library in order to solve all the detected limitations of this library. This new library complements the functionality of the existing primitives to achieve the best productivity for parallel programming with UPC for both expert and novice developers. Additionally, this library has been used to implement another programmability-oriented major improvement for UPC: a MapReduce framework for massively parallel data processing in high performance applications. The development of these new functionalities has been assessed through the use of a wide variety of codes, from small computational kernels to large applications, in order to prove their benefits on different test cases. An important contribution has been the parallelization of a Brownian dynamics simulation with hydrodynamic interactions, evaluated comparatively against its counterpart parallel implementations using MPI and OpenMP, which have also been developed by the author of this Thesis.

Contributions

The main contributions of this PhD Thesis are:

- The detailed description of the UPC language as an alternative to traditional parallel programming solutions (MPI and OpenMP) for code development on hybrid shared/distributed memory architectures. The analysis of benefits and drawbacks against other approaches, specially focusing on the study of the PGAS paradigm, points out the easy adaptation of UPC to these architectures.

- The performance analysis of the UPC language through the microbenchmarking of collective functions and the study of several UPC kernels and applications. The use of different cluster and supercomputer environments helps to provide a wide variety of results with a high significativity.

- The study of UPC in terms of programmability and productivity by means of an empirical analysis with expert and novice parallel programmers. The

evaluation of the codes developed in an isolated and controlled test environment, as well as the feedback about their experience with UPC reported by the programmers, provides valuable information about the potential of UPC and the most relevant areas of improvement.

- The development of a library of extended collective functions, which provides a simple and flexible syntax for the implementation of data movements with UPC in a more productive way. These functions are developed using ANSI C and standard UPC constructs in order to provide complete portability, and they also implement different configurable algorithms that optimize their performance in hybrid shared/distributed memory environments, providing high scalability for large message sizes with a growing number of threads. Moreover, some collective functions executed on the same array (*in-place*) obtain a notably better performance than other traditional parallel implementations such as MPI, confirming the suitability of the functions for HPC codes.

- The implementation of a UPC MapReduce framework for the massively parallel processing of data. The abstraction of communications and the generic data processing allows its integration in any existing code using either sequential C or parallel UPC as base language. Additionally, the use of extended collectives can be enabled to implement the required communications at the "Reduce" stage transparently to the user, thus favoring the efficiency of the framework. A set of applications from information retrieval and image processing fields are used to test the efficiency of the UPC MapReduce framework against analogous approaches on shared and distributed memory, using threads and MPI respectively.

- The evaluation of the new functionalities developed for UPC through the design, implementation and performance analysis of parallel scientific codes. A main contribution is the development of a parallel simulation of the Brownian motion of particles in a fluid: the analysis of dependencies in the code, the adaptation of the code structure to parallel execution environments and the use of extended collectives produce an efficient UPC implementation, which obtains competitive performance when compared to analogous implementations using MPI and OpenMP (also developed in the framework of this Thesis).

Overview of the Contents

This Thesis report is organized into five chapters, whose contents are summarized in the following paragraphs.

Chapter 1, *Parallel Programming with Unified Parallel C*, is intended to give an introduction to the UPC language, presenting the motivations for developing this new PGAS language, based on its differential features versus other parallel programming approaches. Thus, the chapter also covers the description of the foundations of PGAS. Furthermore, different compiler and runtime implementations are described, alongside with a baseline analysis about its programmability, which will determine the next chapter.

Chapter 2, *UPC Productivity Analysis*, focuses on describing the UPC language both in terms of programmability and performance, presenting the guidelines for the subsequent developments in this Thesis. First, an analytical description of the programmability problems in UPC is given, considering the main constructs provided by the UPC standard specification and also giving some relevant performance information. This analysis has been focused on privatizations of shared variables and collective functions. These initial tasks have been complemented with the development of two empirical studies of programmability accomplished on controlled environments (classroom studies). In these studies a group of programmers are asked to develop several parallel UPC codes in a given period of time, and at the end of the session the impressions about the programming tasks are collected by means of a survey. The outcome of the previous analytical and empirical evaluations has been the development of new UPC functionalities in this Thesis, presented in the subsequent chapters.

Chapter 3, *Design and Implementation of Extended Collectives in UPC*, presents the development process of an extended collectives library for UPC, that broadens the applicability of the eight original functions in the UPC specification. First, the main drawbacks of UPC collectives are identified, designing a solution to overcome them through the provision of a set of suitable extended functions. As a result, the library contains a significantly higher number of functions, which can be grouped in four categories: in-place (functions that use the same array as source and destination of communications), vector-variant (supporting the use of different data sizes

for communications in each thread), team-based (collectives that are executed only by a subset of all available threads in a UPC program) and get-put-priv (use of private arrays as source and/or destination). The library includes efficient algorithms based on optimized memory management, which minimizes the allocation and use of buffers for communications, and also supports adaptable executions for multicore based architectures through structured communications using binomial or flat trees (or a combination of both). The performance of the library is assessed using representative kernels and applications (e.g., matrix multiplication and 3D FFT), which are able to benefit from the use of several extended collectives.

Chapter 4, *MapReduce for UPC: UPC-MR*, discusses the implementation of the MapReduce programming model in UPC, a key contribution to improve its productivity in scalable data analytics. The MapReduce processing is based on the application of a function to a set of input elements ("map"), and subsequently a combination of the results to generate a single output ("reduce"). Nowadays this paradigm is mainly used in distributed applications written in Java or C++, but here an HPC-oriented implementation is provided, exploiting UPC facilities to obtain high scalability on multicore systems. The basic features of this framework are its generality (the ability of dealing with any kind of input elements) and simplicity (usable and flexible management functions), as well as being based on an entirely sequential approach to the user, thus abstracting all communications. In order to perform efficient communications between threads at the "reduce" stage, the previously developed extended collectives have been used whenever possible. The developed MapReduce framework has been assessed using representative codes from data processing (e.g., information retrieval and image processing), confirming its high performance and suitability for productive programming.

Chapter 5, *Parallel Simulation of Brownian Dynamics*, presents the parallelization of the simulation of a set of particles using Brownian dynamics with hydrodynamic interactions. This simulation is a representative code in the areas of physics and biology, and shows an exponential algorithmic complexity with multiple data dependencies. As a result of this, different approaches have been followed for its parallelization with UPC, considering the load balancing and the efficient management of communications as key factors, as well as the use of several extended collectives. Furthermore, two additional parallelizations of this Brownian dynamics simulation,

with MPI and OpenMP, have been developed and considered for this work in order to assess comparatively the capabilities of UPC. The evaluation of these parallel implementations confirms the suitability of UPC to provide a productive solution, not only in terms of programmability, allowing a fast time-to-solution, but also in performance on multicore systems, both for shared and distributed memory communications, with significantly high scalability.

Finally, the *Conclusions* chapter summarizes the contributions of this work, confirming that UPC can successfully achieve high programmability and scalability on multicore systems, and pointing out the potential integration of the developed libraries into the UPC specification. A complete description of the functions in the extended collectives library is also included afterwards as an appendix.

Chapter 1

Parallel Programming with Unified Parallel C

This chapter presents Unified Parallel C (UPC) [21], a parallel programming language designed to increase the productivity of the development of High Performance Computing (HPC) parallel codes, especially suitable for hybrid shared/distributed memory architectures. First, the basis of the language and its main features are commented, as well as the state of the art of its implementations, compilers and runtimes. After that, the language is presented in the context of the programmability and productivity requirements of current parallel programming.

1.1. Introduction to UPC

The increasing popularity of multicore systems demands an efficient support for the development of applications on them. The mitigation of single-core processor problems (power, memory and instruction-level parallelism walls) provided by multicore systems has however raised the programmability wall, where developers without parallel programming skills have to confront the growing complexity of new systems. The use of traditional parallel programming models for HPC, such as message-passing, data parallel and shared memory, have been successfully applied on distributed and shared memory environments. However, the hybrid shared/dis-

tributed memory architecture of clusters of multicore nodes presents challenges to the traditional models when looking for the exploitation of all their possibilities in a simple and efficient way, avoiding the cumbersome process of adding parallel programming facilities in a sequential programming model.

As a result of this, significant research efforts have been put in the development of more productive parallel programming paradigms [8], such as the Partitioned Global Address Space (PGAS), which is grabbing the attention of developers of parallel applications looking for programmability. This paradigm considers the existence of two different memory spaces: (1) a private space, in which each thread can define variables that are only accessible by it, and (2) a shared space, that allows communication among threads. The shared space is partitioned between threads, and each partition is said to have affinity to one of the threads in the program, which benefits from data locality in memory accesses to its associated shared space. Thus, the PGAS programming model provides a shared memory view that simplifies code development while it can take advantage of the scalability of distributed memory architectures through the efficient exploitation of data locality.

In the last years several PGAS-based languages have been developed in order to improve parallel programming productivity. Two main approaches have been followed at this point: (1) the earliest one has been focused on taking advantage of existing programming languages in order to extend their functionality implementing PGAS features on them, whereas (2) a more recent approach proposes the development of new languages that provide a suitable set of constructs for parallel programming designed to increase the productivity of code development. The most representative languages of the first branch are UPC (an extension of ANSI C), Titanium [87] (an extension of Java) and Co-array Fortran [9] (which implements co-arrays on top of Fortran). Regarding the second approach, the most relevant achievements are related to the High Productivity Computer Systems (HPCS) project [35], funded by DARPA, which led to the proposal of three languages [43], being each of them developed by a different vendor: (1) X10 [99], a Java-like language developed by IBM in order to be integrated in its own solution for large-scale computing systems, providing data storage divided in "places", which are computational contexts in which different tasks are performed; (2) Chapel [84], a C-like language developed by Cray Inc., which is based on different abstractions that can

support task and nested parallelism, as well as facilities for generic programming; and (3) Fortress [65], a Fortran-like language initially developed by Sun Microsystems (now Oracle Corporation), which supports implicit parallelism and the use of a syntax similar to mathematical formulae.

The adoption and popularity of the languages of the first approach (UPC, Titanium and Co-array Fortran) is mainly motivated by the similar syntax to the base languages, which has favored the development of a significant community of users and developers of each language. The languages that follow the second approach (X10, Chapel and Fortress) have been designed to improve the programmability of the previous ones, but they propose substantial differences with traditional programming languages, which requires the adaptation of the programmers to new environments and structures. As a consequence of this limitation, currently these programmability-oriented languages have not been widely adopted yet and their development is not mature enough (in the case of Fortress, it is being wound down). Thus, improving the programmability facilities of PGAS languages based on the first approach represents a convenient choice in order to put together the best features of both approaches. Bearing this motivation in mind, this Thesis has been conceived to work on the languages of the first approach, and more specifically with UPC, in order to provide new functionalities that facilitate a more productive development of parallel applications using UPC.

1.2. UPC Language and Specification

Different research efforts have pursued the definition and development of the necessary constructions for the UPC language, receiving continuous feedback during the last years by the growing UPC programmers community [96]. In order to coordinate this process, the UPC Consortium has been constituted as the controlling body for the language. It is composed by members of industry (HP, Intrepid, IBM, Cray), academia (University of California at Berkeley, Michigan Technological University [95], University of Florida [91]) and several USA national laboratories and departments. The first UPC language specifications agreed by the UPC Consortium were released in 2001 (v1.0), some modifications were published in 2003 (v1.1), and the latest specifications have been established in 2005 (v1.2).

The main statement of the language definition is that any given C code is also a UPC code, because C variables are considered as private according to the PGAS memory model, and thus independent executions are performed for each thread. Shared variables in UPC are defined with the `shared` keyword before the type declaration of the variable. In order to illustrate the PGAS paradigm in UPC, Figure 1.1 presents a scheme of the memory partitioning for UPC in a four-thread scenario with different variable definitions: a private integer (`i`), a shared integer (`j`) and two shared arrays (`k` and `l`).

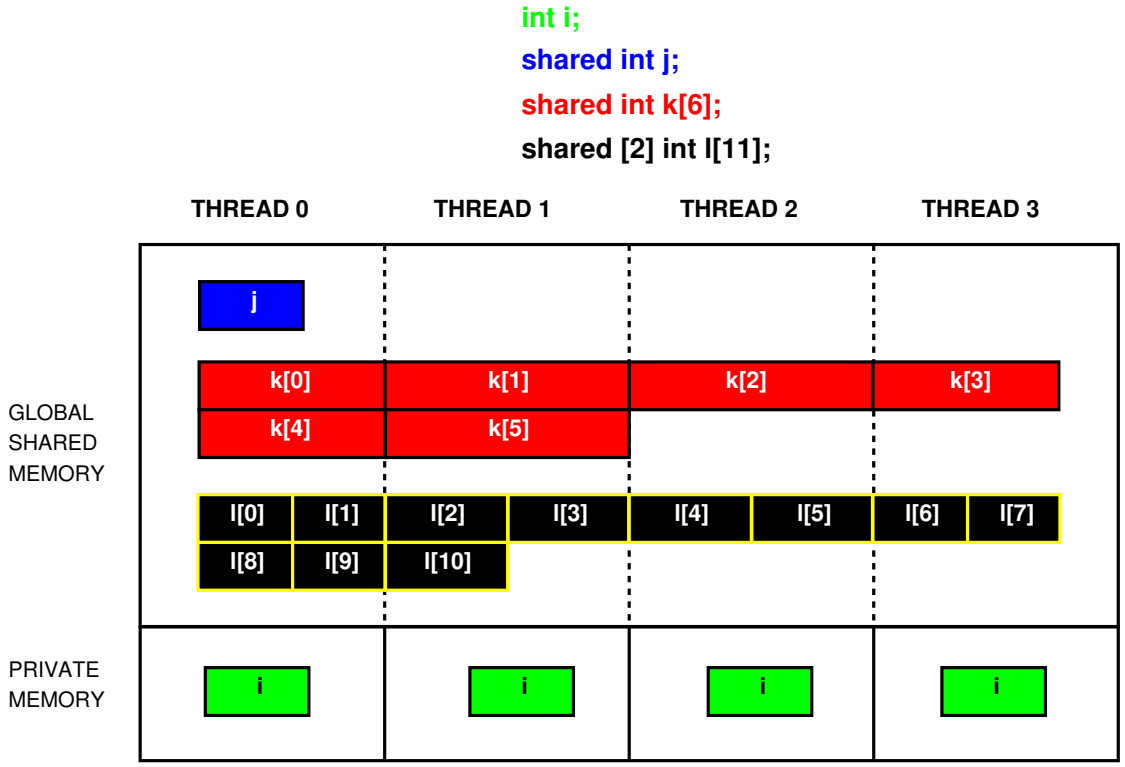


Fig. 1.1: Scheme of the PGAS memory model in UPC

A UPC variable declared as private (single-valued or array) is allocated by every thread in the program in its associated private memory space. If a single-valued variable is declared as shared, UPC allocates memory for it only in the shared space with affinity to thread 0, and thus the variable becomes accessible to all threads. Regarding shared arrays, the elements are stored in the shared space with a cyclic distribution of elements in blocks of a given size. By default, the block size is 1, but a custom block size value can be defined between square brackets just after

the `shared` keyword. If the array length is larger than the product of the number of threads by the block size, more than one block will be assigned to one or more threads following a cyclic approach.

1.3. UPC Compiler/Runtime Projects

Since the definition of the first UPC specification, several compliant implementations of the UPC language have been developed. Here the most relevant projects for the implementation of compilers and runtimes are presented, alongside with additional information on other related UPC projects.

1.3.1. Open Source Projects

The Berkeley UPC (BUPC) distribution [3] is the most relevant open-source implementation of UPC, and currently serves as the main reference for performance evaluations and application development with UPC. BUPC is part of a joint project of the University of California at Berkeley and the Lawrence Berkeley National Laboratory, and it consists of a source-to-source UPC-to-C compiler, which is built on top of an underlying C compiler (e.g., GCC or the Intel C compiler), and a runtime environment, which uses a low-level networking layer called GASNet (Global-Address Space Networking) [88]. This layer is implemented using Active Messages and has direct control over the network architecture. The use of GASNet provides a common environment for PGAS-based programming languages (not only UPC, but also Titanium, for example) by means of different functions (including high-level constructs such as remote data movements and collective operations) that are able to interact with standard and proprietary networks (e.g., InfiniBand, Quadrics or Myrinet, as well as with the Portals API used by Cray XT supercomputers). The current version of BUPC is 2.16.0, which was released in October 2012.

Another outstanding project is the GNU UPC compiler [39], which is a UPC-to-C compiler developed by Intrepid Technology Inc. and licensed under GPL version 3 (2007) that extends the GNU GCC compiler. It is fully compliant with the latest UPC specification, supports several platforms (e.g., Intel, MIPS, Cray), and also

includes a specific debugger for UPC applications (GDB UPC). Nevertheless, GNU UPC is restricted to intranode communications on shared memory, and distributed memory executions are only possible by using its interoperability with the BUPC runtime. One major goal of the project is its integration into the development trunk of GCC, in order to provide UPC support for the current and upcoming versions of GCC.

1.3.2. Vendor Projects

The first fully-compliant commercial implementation of UPC was developed by Compaq (now Hewlett-Packard Company), which has been involved in the development of a UPC compiler since 2002, when Compaq UPC 2.0 was developed. Now, the HP UPC [34] compiler has also been the first commercial UPC implementation to give full support to the current specification in 2005 (under the name "Compaq UPC"). Some other research institutions have collaborated in the development of HP UPC, such as the Michigan Technological University, which already had some experience in the production of a reference runtime environment for UPC [53]. Currently, the latest release of HP UPC is v3.3, which includes VAST optimizations provided by Crescent Bay Software (since v3.0), and supports a variety of systems, mainly focusing on optimizing communications on Infiniband and shared memory on Linux systems.

Cray Inc. has also been involved in the development of its own UPC compiler, Cray UPC [11], which is embedded in the Cray C compiler. It focuses on optimizing UPC communications for Cray supercomputers, and also provides compatibility with different parallel programming libraries. However, Cray UPC imposes some limitations with respect to the UPC specification, which are related to the declaration of shared array dimensions and block sizes, and also includes some deferred functions and statements.

Some other key players in HPC, such as SGI and IBM, are also developing and providing some UPC facilities for their own systems. In these cases, UPC is typically supported by a source-to-source compiler that is integrated on a suite of performance optimization libraries and tools. For example, the SGI UPC compiler uses directly the MPI runtime environment for UPC.

1.4. Comments on UPC Programmability

As commented in the previous sections, the use of C as base language is one of the main characteristics that favor the adoption of UPC: the reason is that C is among the most popular choices for the development of high performance applications, because of its good performance and wide adoption. Additionally, the use of PGAS features in UPC favors the efficient execution of parallel codes on hybrid shared/distributed memory systems, because of the adaptation of the memory paradigm to these architectures. The data movements on shared memory can be local or remote to a node, but the access to data in the same node can be performed more efficiently, thus exploiting data locality. In both cases, the shared memory view of UPC takes advantage of a set of constructs that are designed to obtain programmability:

- Standard language definitions, such as implicit data transfers in assignments of shared variables, and also the predefined constants *THREADS* (total number of threads in a program) and *MYTHREAD* (identifier of each thread).
- Standard high-level constructs for work distribution between threads (e.g., the `upc_forall` loop definition, that assigns the execution of each iteration in the loop to a thread).
- Libraries that provide different constructs, such as memory copy functions and collective primitives.

One important advantage of UPC with respect to other parallel programming approaches is the extensive use of one-sided communications, i.e. communications in a single direction with an active and a passive peer. The threads involved in these communications are not required to synchronize among them, thus allowing the overlapping between independent communications and computations, and consequently providing better performance and especially higher scalability, as the synchronizations tend to burden performance as the number of threads increases. Additionally, collective primitives allow the custom definition of synchronization points at the beginning and at the end of their execution, which also reduces the overall synchronization overhead.

All these programmability constructs are included in the UPC specification, but the community of UPC programmers has been suggesting possible extensions to these features to provide a higher degree of flexibility. Moreover, the implementation of all these constructs represents a challenging process, because of the difficulties in combining programmability and efficiency. Given these facts, the work in this Thesis is intended to contribute to the improvement of the UPC language by analyzing its capabilities in terms of programmers' productivity, and then providing additional functionalities that will improve its syntax, alongside with several examples of kernels and applications that prove the usability of the new constructs. The following chapters will go through all these points, illustrating the lessons learned from each step in this research and motivating the subsequent decisions taken.

Chapter 2

UPC Productivity Analysis

The present chapter analyzes the UPC language in terms of productivity, considering both the programmability provided by the current specification and the detected limitations, and the tradeoffs between programmability and performance in UPC. The main goal here is to identify the strengths and weaknesses of the UPC language and libraries in its application to HPC codes, detecting some possible areas of improvement for the language and providing solutions to the limitations. Thus, this study has been based not only on research in the language specifications and the analysis of the language constructs, but also on experimental information. More precisely, the experience gathered from the UPC community has been contrasted against the results obtained from two experimental evaluations (frequently called programmability sessions or classroom studies), which test programmers' ability to develop UPC codes in a controlled environment. These actions have been the main sources of analytical and empirical feedback that has guided the future developments in this Thesis, with the goal of improving the productivity of UPC.

The organization of this chapter is as follows. First, the most outstanding recent advances on programmability analysis in the area of HPC are commented, mainly focusing on PGAS languages. Then, UPC is analyzed in terms of productivity, considering its programmability according to the current language specification, commenting the demands of the UPC community in this area, and its implications on performance. Next, an empirical programmability study with groups of programmers is described, and the results are analyzed and linked to the study of the

UPC syntax, giving out a complete image of UPC programmability with some additional performance information. Finally, the conclusions argue for the need of improvements on some specific areas, namely the collectives library.

2.1. Programmability vs. Performance with UPC

In general, the studies of programmability in parallel programming have usually been devoted to general considerations and requirements for a parallel language in terms of productivity [43], as well as comments about benefits and disadvantages of the most popular approaches (MPI [52] and OpenMP [85]). Moreover, there are also some additional works on programmability in HPC that propose and analyze different metrics [41, 74] and the design of specific benchmarks [31]. An important conclusion extracted from some of these studies is that a language is considered to be good in terms of programmability if it contains expressive constructs which allow a more compact and simple coding, hence making low-level programming complexities transparent to the user.

Nevertheless, parallel programming languages are difficult to compare accurately in terms of productivity. The main reason for this does not only lie in significant syntax differences between them, but also in the popularity and implantation of the languages. For example, the development of programmability-oriented languages can support parallel programming focused on productivity, i.e. favoring a simple and straightforward development. Thus, general concepts, such as tasks, are used to abstract low-level data structures and data distributions. However, in these cases, there are two main requirements for a parallel language when parallelizing HPC codes: (1) the parallel programming constructs and auxiliary libraries have to be flexible and configurable to provide enough expressiveness; and (2) the implementation of the language has to be mature enough to represent an efficient solution. These two conditions are necessary because the acquisition of programming abilities with a new language requires a period of adaptation, which should be as short as possible in order to be a low entry barrier (low adaptation cost). Furthermore, its functionalities should cover all the requirements of developers.

Moreover, the learning curve (i.e. the process in which proficiency with a new

language is acquired) may be steeper when a new paradigm has to be understood and applied. However, implementing a programmability-oriented parallel language extending a well-known language may facilitate a better adaptation of programmers to parallel programming. Thus, regarding current PGAS approaches, the fact that a large number of HPC applications are nowadays written in C can be considered as one of the main motivations to parallelize them with UPC, especially when looking for programmability. Even though this represents a significant advantage, the two previous requirements (language expressiveness and mature implementation) have to be fulfilled to confirm its suitability, and here some points of conflict appear, as there are some constraints in the UPC syntax and its compilers/runtimes still present some inefficiencies. At this point, the most relevant previous works on programmability have dealt with constructs and algorithms focused on performance increase [57, 101], whereas other approaches tackle the measurement of programming effort in terms of Lines Of Code (LOC) [6]. In general, there are very few specific works on programmability, because the research on UPC has mainly focused on performance evaluation and optimization [10, 19, 20, 104], one of the main shortcomings of initial UPC implementations.

A shared outcome from many performance studies is that UPC shows better performance when data locality and the use of private memory are maximized, establishing some hints as useful programming practices when looking for efficiency [22]. Nevertheless, most of these performance benefits have turned out to be drawbacks when looking for programmability and productivity. Two of these main points of conflict between programmability and performance are shown in the next subsections.

2.1.1. Privatization of Shared Variables

The privatization of shared data with affinity to a thread is a common workaround in parallel programming with UPC, and it is related to the definition of shared memory variables and arrays in UPC. The content of a pointer to shared memory in UPC is not defined by the specification, and therefore each compiler may implement it in a different way, but three concepts are defined for all cases: (1) the thread to which the shared memory pointer belongs, (2) the phase of the pointer, i.e. the offset

of the memory position with respect to the beginning of its corresponding affinity block, which will be nonzero for most elements of shared arrays with definite block size; and (3) the virtual address of the referenced variable in the shared memory space. As a result of this, shared pointer arithmetic presents higher complexity than private pointers (which are equal to ANSI C pointers), and therefore the use of systematic references to shared variables involves a certain overhead that may become significant for a certain amount of processing. In order to illustrate the relationship between this arithmetic and its impact on performance and programmability, Listing 2.1 presents a simple code in which every thread calculates the sum of the values in the shared array `x` with affinity to it. The most straightforward implementation uses `upc_forall` in order to distribute the execution of iterations of the loop between threads, using a pointer to each element of `x` as the affinity parameter: this code takes advantage of UPC programmability facilities and its expressiveness is high. Nevertheless, this implementation uses as many shared memory references as the number of elements in `x`, thus the shared address translations commented before may affect performance for large array sizes.

```
// Common variables: let constant 'N' be a multiple of 'THREADS'
#define N ...
int i, accum;
shared [N/THREADS] int x[N];

// Straightforward implementation using upc_forall
upc_forall (i=0; i<N; i++; &x[i]) accum += x[i];

// Alternative efficient implementation using for
int *x_local = (int *)x[MYTHREAD*N/THREADS];
for (i=0; i<N/THREADS; i++)
    accum += x_local[i];
```

List. 2.1: Example of programmability vs. performance in UPC

The alternative algorithm at the end of Listing 2.1 uses the privatization technique: a pointer to private memory (`x_local`) is used to reference the first element of the shared data block with affinity to each thread. Once the translation of the base address is made, the rest of elements in the array can be dereferenced locally using `x_local`, thus `upc_forall` can be substituted by a `for` loop with a modified

termination condition (here the number of iterations is the number of elements in each data block, instead of the number of elements in `x`). The resulting code is more efficient than the use of `upc_forall`, but it introduces significant complexity, which is not desirable for programmability's sake.

2.1.2. UPC Standard Collective Operations

A traditional controversy between performance and programmability in UPC resides in the language libraries, and particularly in collective communications. The UPC standard collectives library [92], which is part of the UPC standard specification [93], includes eight functions that perform different data-movement (e.g., broadcast and scatter) and computational operations (such as reductions) which are typically used in traditional parallel programming approaches (e.g., MPI). Listing 2.2 presents the signatures of all UPC collective primitives. Despite the fact that they are included in the specification, UPC collectives have not become very popular because of two main reasons: (1) the traditionally low performance of many of these functions, which has led programmers to replace them by combinations of bulk data copy functions (namely `upc_memcpy`, `upc_memget` and `upc_memput`) with additional logic for efficiency purposes, although at the cost of increasing programming complexity; and (2) some limitations for their use which restrict their applicability. Thus, for instance, source and destination arrays must be different and stored in shared memory, and additionally the amount of data per thread involved in the operation should be the same.

In order to analyze the first causes of the poor performance of UPC collectives, an up-to-date analysis of different implementations of collectives was necessary to assess their performance. The results of this performance evaluation have been published in [77], presenting the comparison of multiple implementations of UPC collective communications, as well as representative MPI collective results. MPI is a traditional and widely used approach for parallel programming, and nowadays is the *de facto* standard for programming distributed memory architectures, because of the maturity and optimization of the implementations available (e.g., MPICH2 and Open MPI), thus representing a good reference for performance evaluations. Regarding the implementation of UPC collectives, two libraries have been evalu-

```
void upc_all_broadcast ( shared void *dst, shared const void *src,
  size_t nbytes, upc_flag_t flags
);
void upc_all_scatter ( shared void *dst, shared const void *src,
  size_t nbytes, upc_flag_t flags
);
void upc_all_gather ( shared void *dst, shared const void *src,
  size_t nbytes, upc_flag_t flags
);
void upc_all_gather_all ( shared void *dst, shared const void *src,
  size_t nbytes, upc_flag_t flags
);
void upc_all_exchange ( shared void *dst, shared const void *src,
  size_t nbytes, upc_flag_t flags
);
void upc_all_permute ( shared void *dst, shared const void *src,
  shared const int *perm, size_t nbytes, upc_flag_t flags
);
void upc_all_reduceT ( shared void *dst, shared const void *src,
  upc_op_t op, size_t nelems, size_t blk_size,
  TYPE(*func)(TYPE, TYPE), upc_flag_t flags
);
void upc_all_prefix_reduceT ( shared void *dst,
  shared const void *src, upc_op_t op, size_t nelems,
  size_t blk_size, TYPE(*func)(TYPE, TYPE), upc_flag_t flags
);
```

List. 2.2: Signatures of the UPC collectives included in the current specification

ated: (1) the collectives implementation provided by the Berkeley UPC distribution (from now on, Berkeley Collectives or BCOL) [3], and (2) the reference implementation of UPC collectives based on standard UPC memory copy functions, which was developed at Michigan Tech. University (from now on REF) [54].

BCOL is based on the low-level GASNet communication library [88] implemented on top of Active Messages [97]. From version 2.6.0 of the BUPC compiler, the former linear flat-tree algorithm implementation of collectives has been replaced by a binomial-tree communication pattern, which organizes data transfers in a logarithmic number of steps, reducing memory and network contention. The current implementation of collectives has remained without major changes from v2.6.0 up to the latest version (v2.16.0). Regarding REF, the implementation of its collective primitives is based on standard UPC `upc_memcpy` data transfers. Its communications use a fully parallel flat-tree algorithm, performing all communications in parallel in

a single step. For illustrative purposes, Figure 2.1 presents the differences between communications using a flat tree (left) and a binomial tree (right).

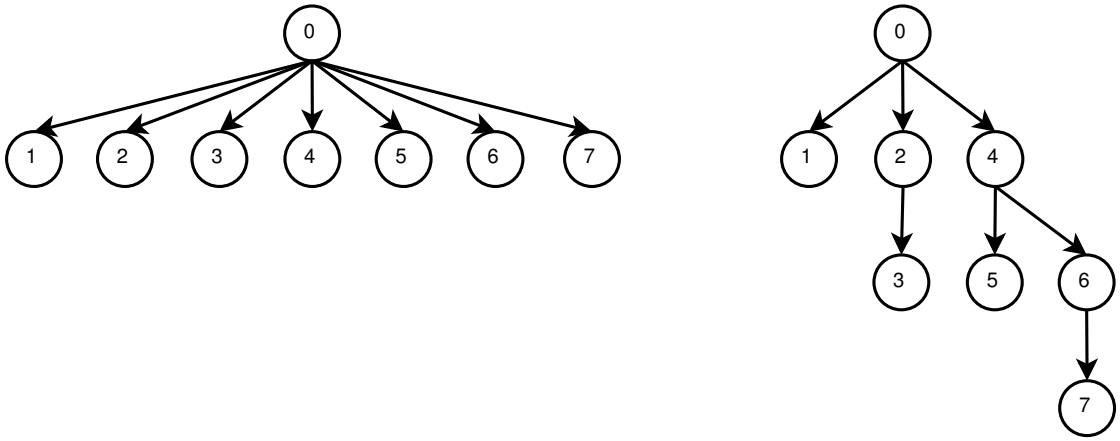


Fig. 2.1: Collective communications: flat tree vs. binomial tree (8 threads)

Table 2.1 shows the relative performance from the microbenchmarking of a set of representative UPC collectives (broadcast, scatter and exchange) compared to their MPI counterparts (`MPI_Bcast`, `MPI_Scatter` and `MPI_Alltoall`, respectively) using 32 threads either on 4 HP rx7640 nodes or 1 HP Superdome node of the Finis Terrae supercomputer at the Galicia Supercomputing Center (CESGA) [25]. Each HP rx7640 node has 8 Montvale Itanium2 (IA-64) dual-core processors at 1.6 GHz, 128 GB of memory and InfiniBand [38] as interconnection network, whereas the HP Superdome has 64 Montvale Itanium2 processors (128 cores) at 1.6 GHz and 1 TB of shared memory. Berkeley UPC v2.6.0, with Intel C compiler 10.1 as backend and the IBV conduit (InfiniBand Verbs) activated for GASNet communications, was used for the UPC microbenchmarking, whereas HP MPI v.2.2.5.1 was selected as MPI implementation as it showed the highest performance in this system. Shared memory results ("SMP") have been obtained from the Superdome, whereas pure distributed memory measurements with MPI, as well as UPC on InfiniBand ("DMP") and hybrid shared/distributed memory ("HYB"), have been obtained from the HP rx7640 nodes. Both DMP and HYB use 8 threads per node, so the difference between both measurements lies in the fact that for HYB the intranode communications take advantage of shared memory transfers, whereas for DMP GASNet transfers are used for intranode communications. These results have been obtained for two representative message sizes (1 KB and 1 MB), and the throughput of UPC collectives is

shown as a percentage of the MPI performance, thus UPC outperforms MPI when the percentage is higher than 100%. This situation, UPC outperforming MPI, only happens when communicating 1 MB messages on REF SMP, because of the high performance of the parallel access in the shared memory of the destination threads to the data of the source thread, without any additional synchronization.

As BCOL on SMP implements a binomial-tree algorithm, it needs five steps ($log_2 32$) to implement the operations, thus obtaining poorer performance than REF because of the synchronization overhead incurred by each of the five steps, compared to the single step synchronization required for REF. Regarding the HYB microbenchmarking scenario, UPC performance is lower than MPI results. Furthermore, UPC suffers from higher start-up latencies than MPI, which means poor performance for 1 KB messages, especially for the broadcast. This comparative analysis of MPI and UPC collectives has served to assess that there is room for improvement in the implementation of the UPC collectives by taking advantage of shared memory communications. The traditional inefficiency of UPC collectives is mainly restricted to distributed memory communications, and this overhead in the execution times may be lightened by implementing efficient algorithms that exploit data locality on multicore systems according to the required communications between threads.

	Broadcast		Scatter		Exchange/Alltoall	
Library \ Message size	1 KB	1 MB	1 KB	1 MB	1 KB	1 MB
MPI (GBps)	0.0992	4.4013	0.0088	1.5360	0.0066	0.0971
BCOL SMP	3%	22%	23%	44%	21%	40%
REF SMP	8%	**145%**	61%	**171%**	82%	**514%**
BCOL DMP	2%	30%	13%	28%	11%	53%
REF DMP	1%	15%	11%	40%	6%	52%
BCOL HYB	5%	86%	43%	44%	16%	89%
REF HYB	9%	24%	72%	97%	13%	68%

Table 2.1: UPC vs. MPI collectives performance (32 threads/processes, MPI = 100%)

Considering the second reason for the lack of adoption of UPC collectives (the limitations in their applicability), little research has been conducted in this area. Up to now, the most relevant proposals on extensions for UPC collectives were described in a technical report [68] and in a draft specification [70] elaborated at Michigan Tech. University (MTU). These documents propose the implementation

of several collective extensions using concepts already present in other parallel programming languages and libraries (e.g., MPI), such as the definition of variable-sized data blocks for communications (vector-variant collectives), a simplified interface for communications in the same shared array (in-place collectives), the use of teams (subsets of the threads that execute a UPC program), and also asynchronous data transfers. The use of one-sided communications (see Section 1.4) is considered as the main basis to implement collectives [69]. Other related proposals are value-based collectives [4], which use single-valued variables, either shared or private, as source and destination of communications.

However, the vast majority of these works on extended UPC collectives represent just a sketch on how these collectives could be implemented. In fact, only the MTU report [68] presents some implementation details and a preliminary benchmarking for a small subset of these functions, whereas many other issues, such as the implementation of teams or in-place operations, are simply mentioned, without further discussion. The main research efforts on UPC collectives have traditionally focused on performance analysis and the proposal of potential performance optimizations to the standard collectives library [71] and low-level tuning support in order to build a more efficient library [58]. This latter work also comments some hints about the use of routines that would support handling subsets of threads in a team according to the affinity of the data involved in the collective call, similarly to MPI communicators. Nevertheless, few implementation details are given on this issue, as that work focuses mainly on performance evaluation on multiple systems.

2.2. Empirical Programmability Analysis of UPC with Classroom Studies

Besides the analysis of language features and code development, a good way to prove if a language provides good programmability is to make a survey on a group of programmers. The benefits and disadvantages reported by various code developers, especially when they have different skills, can give valuable information about the programming language and also help to guess if it could become popular among the parallel programming community. Classroom programmability sessions are used as

a good reference in order to measure productivity. Some tests with homogeneous groups of students have also been carried out, obtaining general conclusions for different languages and paradigms [1, 28, 47]. UPC has also been considered for a programmability study [62] in comparison with MPI that includes an analysis of statistical significance of the results. However, the development times were not taken into account and the experimental conditions of the study were undefined. In these studies, the typical measures that are used to evaluate programmability are Source Lines Of Code (SLOC) and speedup, directly measured from the codes developed during a programmability session. Here, special tools are generally used to manage the logs of the test, as well as to report complete information about the work performed by each participant in the study [24]. Among these tools it is worth mentioning UMDInst [15], which consists of a set of wrappers that create XML logs including the most relevant actions performed at code development (edition, compilation and execution), also saving snapshots of the codes. These features help to give a more accurate measure of the development time and cost associated to the parallelization of code.

This section presents the results of two classroom studies [83], each one consisting of a four-hour session with a group of UPC-inexperienced programmers. These studies have been carried out in the framework of the research of this Thesis, in its early stages (February 2009) in order to obtain useful information to motivate further actions. In the first session, the participants are final-year students of the B.S. in Computer Science at the University of A Coruña (UDC) [90], whereas in the second one the participants are a heterogeneous group of research staff at the Galicia Supercomputing Center (CESGA) [29]. These two groups present special features that are interesting for the analysis, specially in terms of programmer profile. The students at UDC are a quite homogeneous group, with minor variations in their academic curricula, but the staff at CESGA present clearly different profiles, as they have different degrees, specializations and work experience.

The development of these sessions is structured in different stages. First, the participants fill out a form to characterize their profile. Then, a seminar explaining the basic constructs of UPC (using a slide show and some practical examples) is given to the programmers. Afterwards they are asked to parallelize several sequential codes in order to test the acquired skills. Finally data about their impressions

on UPC and some detected benefits and disadvantages are obtained. The main advantages of this approach are: (1) the time control of the development stage, (2) the use of inexperienced programmers in the UPC language but with some general background knowledge on different programming paradigms, and (3) the inclusion of their opinions as a complement to the analysis of the developed codes, which gives some guidelines to identify desirable features in a parallel programming language.

The next subsections present the details of this programmability study. First, the design of the activities is outlined, presenting the codes and software used in the sessions. Afterwards, detailed information about the most relevant results is given.

2.2.1. Design of the Programmability Study

As commented before, the two sessions organized at UDC and CESGA have followed the same overall structure. Initially, each participant was asked to fill out an initial questionnaire about his/her academic profile and parallel programming background, as well as his/her interest on this area. The questions have been slightly adapted to the participants of each study. After this initial questionnaire, the participants attended a seminar on UPC. The contents of this talk were taken from slides used in UPC seminars at UC Berkeley [100], and included all basic concepts and constructs needed to understand the PGAS paradigm (shared address space vs. private address space) and develop UPC codes (`upc_forall` construct, barrier synchronizations, pointers, distribution of shared arrays and raw memory copies). During the presentation of this material, the students were asked to test some sample codes ("Hello World", Pi computation using the Monte Carlo approach and matrix-vector multiplication). The UPC seminar, including the explanations and the execution of the test codes, had a duration of about 1.5 hours.

After that began the development stage, where the participants were asked to develop three parallel codes in UPC from their sequential versions, implemented in C, that were given to them. The overall coding time was set to be 2 hours. The three proposed codes were:

- A simple Stencil operation on a 10^6-element vector, analogous to one of the example codes included in the Berkeley UPC distribution. The Stencil operation

over the 10^6 elements of the vector is performed 100 times (it uses an external 100-iteration loop). The initial sequential code had 21 SLOCs in total.

- The Buffon-Laplace Needle problem, a Monte Carlo simulation that gives an accurate approximation of Pi based on the probability that a needle of length l that is dropped in a grid of equally spaced parallel lines will touch at least one line. The number of predefined trials is 10^7. This sequential code had 90 SLOCs.

- The Computation of the Minimum Distance among different nodes in a graph (a version of the Floyd-Warshall algorithm [27]). This code was implemented by some of the students at UDC as an MPI project during a previous course on parallel programming. The size of the source distance matrix is 500x500 (that is, 500 nodes). This code had 63 SLOCs.

After parallelizing each code, the students were asked to run their codes and then report their performance. Finally, the participants had to fill out a final survey about their impressions about UPC, their interest on the language, the benefits or disadvantages they could notice and the features they would like to see in UPC.

The number of participants that attended the classroom study at UDC were 22, all with some previous knowledge of MPI and OpenMP. At CESGA, 13 programmers with different profiles (e.g., B.S. and PhD degrees in computer science, physics and mathematics) took part in the experiment. In general, all participants at CESGA had previous experience with programming languages, but only a few reported to have a previous knowledge on parallel programming. Even some of these programmers did not have much experience with C, as they were used to programming with other languages (e.g., Java and PHP).

The two experimental testbeds used for the execution of the developed codes were: (1) 8 nodes of an InfiniBand cluster at UDC [64], with 4 cores per node and Simultaneous Multithreading; and (2) 16 single-core nodes of a Myrinet cluster at CESGA. All performance results were obtained using the Berkeley UPC compiler, version 2.8.0 [3]. The UMDInst system was used to generate logs. A summary of the most relevant results from each code was obtained with two Perl scripts that parsed the UMDInst logs. Additionally, the number of SLOCs for each code was obtained using the CLOC Perl script [60].

2.2.2. Analysis of Results

The analysis of the different results obtained in the two classroom studies has been accomplished using two sources of information: the codes developed by each participant and their profiles and opinions about UPC. The study of the codes is based on the speedup achieved, the number of SLOCs and the development time for each code.

Code	**# Participants**	**Outcome**		
		Correct	**Incorrect**	**N/A**
Stencil (UDC)	22	20	2	0
Stencil (CESGA)	13	10	3	0
Buffon-Laplace N. (UDC)	22	15	7	0
Buffon-Laplace N. (CESGA)	13	9	3	1
Minimum Dist. (UDC)	22	7	8	7
Minimum Dist. (CESGA)	13	2	4	7

Table 2.2: Summary of codes obtained in the classroom studies

Table 2.2 presents a summary of the UPC codes developed in the classroom studies undertaken, indicating the number of correct and incorrect codes developed by the participants. Two conclusions can be extracted from this table: on the one hand, most of the participants were able to obtain a correct solution for the Stencil and the Buffon-Laplace Needle problems; on the other hand, few of them were able to parallelize the Minimum Distance Computation code. Regarding the incorrect implementations produced by some of the participants, there are several points in common among them: the Stencil code is quite simple, but five people (two at UDC and three at CESGA) did not obtain a correct solution, being the most common error the parallelization of the external iterations loop instead of the internal Stencil loop. Although the results obtained with these codes are correct, this work distribution causes all threads to perform all the operations over the whole array instead of splitting the array processing among the different threads. Thus, these participants have misunderstood the basic concepts of SPMD programming and work distribution with `upc_forall`, and hence their implementations for this code are highly inefficient.

The incorrect Buffon-Laplace Needle codes (7 at UDC and 3 at CESGA) also shared a common error: these participants forgot to synchronize the threads after the

computation of the Pi estimation in each thread. This race condition can produce an erroneous result, because there is no guarantee that the partial results are updated when the root thread tries to retrieve them from shared memory. Additionally, 6 out of these 10 erroneous codes also used a scalar shared variable to retrieve without synchronization the partial results for every thread, instead of storing them in a shared array and performing a reduction on it, thus causing a race condition. Again, this error is due to a misunderstanding of the UPC memory model.

Finally, unlike the previous cases, the characterization of the errors in the Minimum Distance code is more difficult, because many of them are due to a bad initialization of the matrix or some misunderstanding of the algorithm. In general, many participants (even the ones that developed a correct code) found it difficult to deal with shared array arguments in functions because of the block size definitions.

The results of each session are next analyzed in more detail. Tables 2.3-2.8 show the results of all correct UPC codes developed in the programmability session at UDC (Tables 2.3-2.5) and CESGA (Tables 2.6-2.8), respectively. The codes have been classified according to the performance obtained in terms of speedup when they are executed with up to 16 threads. The speedup is specified qualitatively, and each of these values corresponds to a different behavior of the speedup for this code when the number of threads increases (e.g., if the efficiency is almost 100%, the speedup is considered as "Excellent"). Alongside the speedup, the average number of SLOCs and the average development time of all the codes included in each group are shown in every table, because they can help give a measure of the acquired capability of each group to develop UPC programs.

Speedup	**Num. Codes**	**Num. SLOCs** (average)	**Development Time** (average)
Excellent	1	28	38' 19"
Quite good	4	22	49' 35"
Good	12	23	50' 28"
Fair	2	22	50' 20"
Bad	1	23	1 h 1' 43"

Table 2.3: Stencil with 10^6 elements (UDC group)

The results presented in Table 2.3 show that there are five possible qualifica-

tions in terms of speedup for the 20 UPC Stencil codes developed in the session at UDC. As the parallelization of this code is defined using a few syntactic constructs, it is easy to find a correlation between the use of these constructs and the speedup obtained by each code. Thus, the only code that included the correct parallel constructs, and also used a privatization method for one of the arrays, could obtain an "Excellent" speedup (which means that it was very close to the ideal). A "Quite good" speedup (about 80% of parallel efficiency) was obtained by codes that included the correct parallel constructs, but without a privatization method (in these cases, they used a `upc_forall` loop with an optimal blocking factor for the arrays and performed all operations in shared memory), thus obtaining a slightly lower speedup than with privatization. Most of the Stencil codes obtained a speedup rating of "Good", because they were parallelized using the `upc_forall` loop, but with non-optimal array blocking for the arrays. A "Fair" speedup was obtained by two codes that implemented the `upc_forall` loop, but performed a particular array blocking: the two arrays used for the Stencil operation had different blocking factors, that is, one had a block distribution and the other used a cyclic distribution. Thus, depending on the definition of affinity in the `upc_forall` loop, these codes achieve a different speedup, but it is always lower than for the previous cases. Finally, one code could not get any speedup at all, because of a bad definition of the affinity in the `upc_forall` loop, which maximized the number of remote accesses.

Additionally, the study of SLOCs for Stencil indicates that a quite good speedup can be obtained without increasing the size of the code, but the best performance is achieved with more lines. This is due to the use of privatization, that requires additional processing (e.g., the definition of private arrays and the copy of the shared arrays to private memory).

In terms of development time, the participants at UDC spent around 50 minutes on average with this code. As stated during the session, many of them had reviewed all the information about UPC that was given to them during the seminar in order to develop their first UPC code, therefore part of this development time can be assigned to a small learning curve. Nevertheless, some significant differences among participants were appreciated here.

Table 2.4 presents the results for the Buffon-Laplace Needle code. All the developed codes achieve excellent speedup. The reason is that Buffon-Laplace Needle

Speedup	**Num. Codes**	**Num. SLOCs** (average)	**Development Time** (average)
Excellent	15	98	28' 43"

Table 2.4: Buffon-Laplace Needle with 10^7 trials (UDC group)

presents few parallel alternatives: a correct parallel code is likely to obtain high speedup, and the lack of a feature in the code (e.g., synchronization barriers, shared array definition) tends to result in an incorrect program. It is also significant that the development time for this second exercise is less than for the first one. This happens because this code is based on the evaluation of trials similarly to the computation of Pi using the Monte Carlo method, which was proposed as an example in the UPC seminar. Thus, many participants probably found the analogy between these two codes and they could obtain a correct code easily. Regarding the length of the codes, the average number of additional SLOCs used here in order to parallelize this code is 8 (the sequential version has 90 SLOCs).

Speedup	**Num. Codes**	**Num. SLOCs** (average)	**Development Time** (average)
Excellent	4	75	1 h 1' 34"
Bad	3	70	46' 26"

Table 2.5: Minimum Distance Computation with 500 nodes (UDC group)

Table 2.5 shows the results of the correct Minimum Distance codes, where the speedups are classified as "Excellent" (4) and "Bad" (3). The use of correct `upc_forall` loops and privatizations of variables is the reason why the best four versions of this code obtained excellent speedup. In terms of SLOCs, it deserves to be mentioned that privatization involves the inclusion of some additional code (in the best four cases, 12 lines on average). Other improvements were implemented with less SLOCs (on average, 6 extra lines were added in different parallelizations using `upc_forall` loops), but the benefits were slightly lower in terms of speedup. The average development time for the best four codes is high (about 1 hour), probably because the correct implementation of privatizations for these codes may have taken longer.

Speedup	**Num. Codes**	**Num. SLOCs** (average)	**Development Time** (average)
Quite good	7	24	40' 19"
Bad	3	22	48' 30"

Table 2.6: Stencil with 10^6 elements (CESGA group)

Table 2.6 presents the speedup qualification of the Stencil codes developed in the classroom study at CESGA. Here there are only two types of codes: the ones that achieved a quite good speedup and the ones that showed bad speedup. The former used a correct `upc_forall` loop and a suitable blocking of arrays, whereas the latter used a wrong parameter of the affinity in `upc_forall`, which did not match the selected array blocking. The average number of SLOCs for these codes (24 and 22 SLOCs, respectively) is very close to the sequential code (21 SLOCs). However, there are significant differences among the codes developed at CESGA, which are due to the different profiles of the participants. The development time of this code is also a bit lower than at UDC, because many participants did not spend too much time testing this code, and some of them had a great ability to quickly develop the required code.

Speedup	**Num. Codes**	**Num. SLOCs** (average)	**Development Time** (average)
Excellent	9	101	46' 20"

Table 2.7: Buffon-Laplace Needle with 10^7 trials (CESGA group)

The results shown in Table 2.7 are analogous to the ones obtained at UDC: all the correct codes achieved the best possible speedup. Nevertheless, there are significant differences in terms of SLOCs and development time, as the participants at CESGA used, on average, more SLOCs and more time to develop the parallel code. Once again, there are noticeable differences (high variability) among CESGA programmers, because many of them did not realize quickly that this program was analogous to the Pi computation presented in the seminar, and therefore they had problems on deciding which was the best strategy to parallelize this code. This led to a number of editions and compilations higher on average than in the UDC session.

However, as for Stencil, some participants could get to parallelize this code quite quickly.

The Minimum Distance code has posed a great challenge to CESGA programmers: this code was not familiar to any of them, unlike for the UDC session. Therefore, its complexity and the time involved in developing a correct solution to the previous codes were the reasons why only half of the participants in the session at CESGA started developing the parallel version of this code. As shown in Table 2.8, none of the two correct codes could get a good speedup. However, it can be seen that one code obtained a slightly better speedup with a higher development time.

Speedup	**Num. Codes**	**Num. SLOCs** (average)	**Development Time** (average)
Bad	1	90	1 h 10' 53"
Very bad	1	66	30' 31"

Table 2.8: Minimum Distance Computation with 500 nodes (CESGA group)

As commented before, some UDC students that attended this session had also developed previously an MPI version of the Minimum Distance code, after a 12-hour course on basic notions of parallel programming (4 hours) and MPI (8 hours). In order to present a comparison between the MPI and UPC codes, Table 2.9 shows the speedups ("+" means "Excellent", "-" means "Bad"), number of SLOCs and development time of pairs of MPI and UPC codes developed by the same students (named S-xx). Five of the seven students that obtained a correct UPC version of this code had previously developed an MPI implementation, thus these five students are the ones included in the table. The MPI development times are an estimation of the time consumed in the study and parallelization of this code, and it was reported by the corresponding student.

The analysis of the MPI and UPC results clearly indicates that UPC allows an easier and faster parallelization than MPI: compared to their MPI codes, three students could get a similar speedup with UPC using less development time and SLOCs (the sequential code has 63 SLOCs). Although MPI time measurements have not been strictly controlled, the development time estimation and the SLOC count suggest that there is a high difference in terms of programmability among both approaches. In fact, other two students that had not developed the MPI version of

ID	Speedup		Num. SLOCs		Development Time	
	MPI	UPC	MPI	UPC	MPI	UPC
S-04	+	-	136	68	24 h	35'
S-05	+	+	139	73	36 h	1 h 21'
S-15	+	-	158	76	15 h	1 h 6'
S-17	+	+	159	87	15 h	1 h 31'
S-20	+	+	174	66	18 h	1 h 24'

Table 2.9: Minimum Distance Computation - MPI vs. UPC

Minimum Distance were able to obtain a correct UPC parallel code during the classroom study (one of them achieving an excellent speedup), which confirms that the time necessary to understand the problem is not very high. Moreover, the learning time for MPI was longer than for UPC, which confirms the effectiveness of UPC for a quick parallelization. However, for most of the students, MPI was their first approach to parallel programming, so this fact has to be taken into account as the MPI learning curve perhaps was larger because of the lack of previous experience in parallel programming. It is also important to note that the average development time of the best UPC codes was near an hour and a half, therefore if the time for the session were longer than the two hours scheduled probably more participants could have developed the UPC version of this code.

2.2.3. Reported Comments on UPC

The comments of the participants of this study on UPC are presented in Figures 2.2 and 2.3. Figure 2.2 indicates that the most important benefit for many of the participants is that UPC is an extension of C, which confirms the previous statement commented in Section 2.1: a good knowledge of C simplifies the development of parallel applications with UPC. Additionally, about half of the answers to the test considered that the UPC memory model is a helpful way to develop codes, thus the use of PGAS is also perceived as a potential advantage for UPC. Some differences among the two groups of participants are stated when asked whether UPC allows easy prototyping: more than 85% of the UDC students consider that UPC facilitates prototyping, whereas less than 40% of the CESGA programmers have reported the

same feature, which is probably due to the different profiles of both groups of participants. More specifically, some CESGA programmers are used to working with distributed applications that use object-oriented languages (e.g., Java), and even though they appreciate the benefits of UPC as a PGAS language based on C, they do not consider the development of prototypes for distributed applications using C, according to their experience.

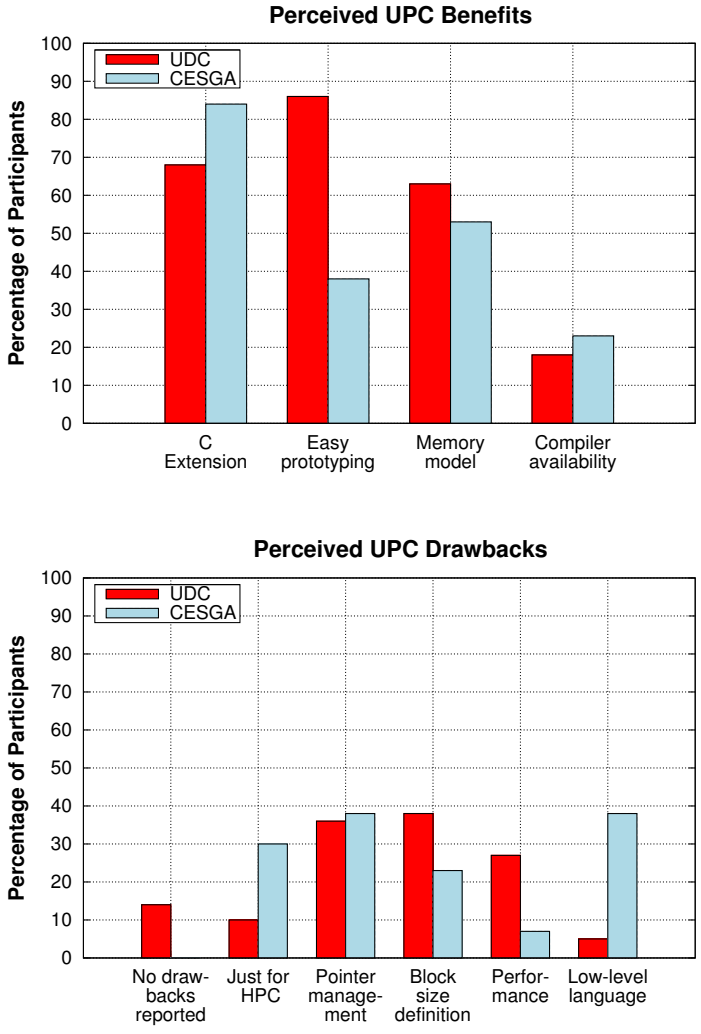


Fig. 2.2: Reported benefits and drawbacks of UPC

The main drawbacks reported in Figure 2.2 (bottom graph) are also different depending on the group of people studied. Both groups agree that one of the main

difficulties in UPC is the management of pointers, which is related to the perception that privatizations are necessary to obtain performance. Although this issue has been reported by less than 40% of both groups, this has been a general perception of most of the participants, which during the sessions have pointed out the need of a compiler-based solution for the implementation of privatizations at low level and transparently to the programmer. Additionally, the general impression at CESGA is that UPC is a quite low-level language that may only be used in HPC, which is related mainly to the unfair comparison with object-oriented languages. Participants at UDC have found that the definition of block sizes was one of the most important drawbacks in UPC, alongside with the perception of poor performance, also related with the use of privatizations.

The opinions about the most urgent issue to solve in UPC, that are shown in Figure 2.3, are similar to the previous ones: 30% of CESGA programmers reported the need for language abstractions that help to obtain simpler codes. In fact, following the same ideas reported in the questionnaire of benefits and drawbacks, many of the CESGA programmers suggested that the language could provide libraries with high-level constructs to abstract the parallelization of the codes. UDC students' complaints have focused on language performance, as well as on general data distribution issues.

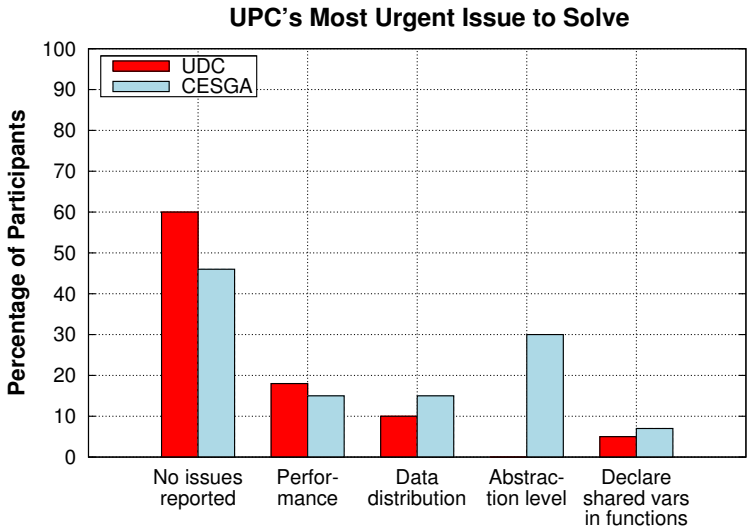


Fig. 2.3: Most urgent reported issue to solve

These figures reveal that there are different opinions reported by UDC and CESGA participants, and the trends extracted from the results shown in each figure are also consistent with the results presented in the rest of the figures. The participants' background knowledge, interests and motivation are determinant in their opinions, but a wide variety of improvement areas can be detected. In general, participants with some specific knowledge on parallel programming have seen some advantages in terms of programmability in the use of UPC, but they have been aware of performance drawbacks, and they reported about the high programming effort required to solve them. Participants with less experience on parallel programming have missed the use of high-level constructs, and consequently their feedback is mainly related to the lack of programmability facilities.

2.3. Proposal of Programmability Improvements

The analysis of the UPC language in terms of programmability has given out several conclusions, which have been confirmed and assessed in the classroom studies developed with different programmers. In general, UPC has shown good potential in order to provide programmability when programming HPC applications, but there are some significant areas that have to be improved, especially regarding the implementation of compilers and runtimes. The exploitation of PGAS features is an important source of flexibility and simplicity for the language, but the use of privatizations has shown to be very important when looking for performance, although it does not favor the development of simple and readable codes. Additionally, the language libraries provide very important features, such as collective communications, but their usability is restricted because of some efficiency problems and their limited applicability, which is always restricted to the PGAS shared memory region and fixed communication sizes. In fact, when applying the privatization on a shared array, the privatized array cannot be used for collective communications with the standard UPC functions.

The detected conflicts between the efficiency and the simplicity of UPC codes, as well as the reported drawbacks of the language in the programmability session, indicate that the UPC specification still needs some more improvements. Currently, the draft for a new specification document (v1.3) [89] is being discussed by the UPC

community, and the major changes proposed by the UPC Consortium will imply the separation between language specifications and library specifications (where libraries can be classified as "required" or "optional"). According to this context, a few language-specific constructs could be included, but currently the main developments of the UPC community should focus on the definition of new libraries and extended versions of the existing ones.

At this point, the development of this Thesis has been conceived to contribute to the development of UPC by promoting and extending the UPC standard collectives library, giving a solution to all the detected limitations. This new extended library will provide the necessary complement of the existing functions to obtain the best programmability facilities in these communications for both expert and novice parallel programmers. The following chapters of this dissertation will focus on presenting the interfaces of different functions in the library, as well as the algorithms and definitions that will facilitate its portable implementation (thus being applicable to any specification-compliant UPC compiler). Very special stress will be put in presenting the usability and applicability of the developed functions, and therefore a wide variety of kernels and applications that can take advantage of the extended collectives will be presented in order to illustrate the benefits of using this extended library in terms of programmability and performance.

Chapter 3

Design and Implementation of Extended Collectives in UPC

The limitations of the UPC collectives library discussed in the previous chapter have motivated the development of a new library of extended UPC collective functions [82]. The main goal of this library is to solve the shortcomings detected in this area of the current standard UPC specification, and thereby improve the programmability and productivity of UPC. The previously presented research on performance and programmability for UPC, backed by the experimental data and the proposals sketched by the UPC community, has been taken as a basis to develop different sets of functions in the library (in-place, vector-variant, team-based), and also new functions (get-put-priv) whose features are also combined with the previous ones (e.g., get-put-priv vector-variant collectives). As a result, the developed library covers most of the demands of the collective communications required by UPC programmers. Moreover, an implementation of UPC teams at library level has been developed to support team-based collectives, thus alleviating the lack of a standard implementation of teams in the UPC language specification. The main contributions of this work are not only the definition of the interfaces and the operation of each function, but also (1) the implementation using standard UPC constructs, thus making the library completely portable to any compliant UPC compiler and runtime, and (2) the design decisions taken to implement some operations using scalable and efficient algorithms, which have been evaluated in terms of performance and pro-

grammability through its application in the development of several representative UPC codes.

The organization of this chapter is as follows. First, Section 3.1 comments the most relevant design decisions and strategies that have driven the development of this library, including its structure. After that, the signatures of the functions and implementation details of the collectives are presented and explained in Section 3.2, discussing different possibilities for their implementation on different computer architectures, and some remarks about their optimization are given in Section 3.3. Section 3.4 presents four kernels used as representative examples of application of these collectives: matrix multiplication, both for dense and sparse computation, Integer Sort and 3D Fast Fourier Transform. Finally, Section 3.5 comments the measured microbenchmarking results of representative collectives, and Section 3.6 shows the performance results of the kernels, analyzing the impact of the use of the developed collectives. Appendix A presents the complete API of the library.

3.1. Design of the Collectives Library

The functions included in the new extended UPC collectives library are distributed in four different groups, each of them focused on overcoming a specific limitation of the standard UPC collectives. Figure 3.1 presents the four main limitations of the standard collectives (left-hand side), and the groups of implemented collective functions (right-hand side) that address the corresponding issue. Thus, these groups of collectives provide different programmability improvements:

- *In-place collectives*: overcome the need of using different buffers for source and destination data.
- *Vector-variant collectives*: allow the communication of a varying data size per thread.
- *Team-based collectives*: execute collective operations within a particular team of threads.
- *Get-put-priv collectives*: skip the limitation of using shared memory addresses as function parameters.

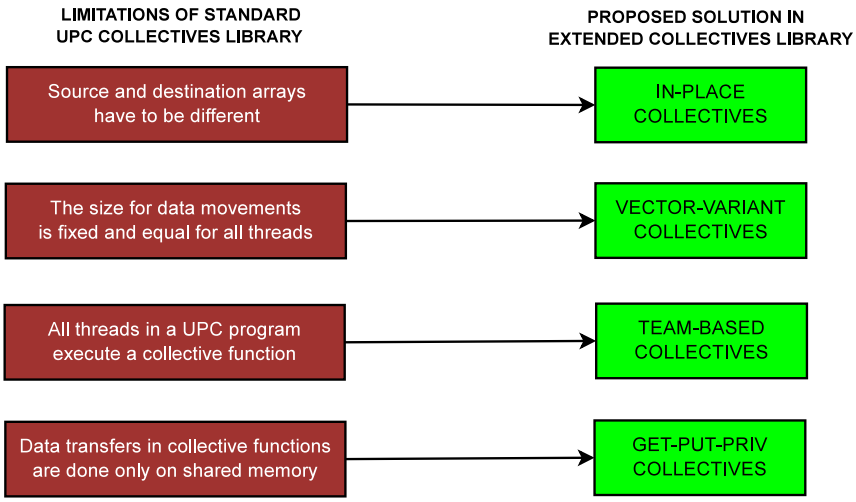


Fig. 3.1: Scheme of the Extended Collectives library

In general, the development of these extended collectives has been guided by the following principles:

- The operations implemented are those present in the standard UPC collectives library (see Listing 2.2 for a full reference). Additionally, the allreduce collective has been included as a subject for extension, because of its usefulness in different applications.

- The arguments of all the extended collectives are always derived from their standard counterparts, and new parameters are added to or removed from their interface in order to implement the required extended functionality. The main goal here is to keep interfaces simple, while providing enough flexibility.

- Additional features can be introduced in some extended collective functions included in this library, providing another collective with specific features. For example, a significant subset of the implemented primitives provides the possibility of defining a specific thread as the root of the communication operation: these new functions are the *rooted* collectives of the developed library. The supported additional features may differ depending on the extended collective.

- The extended collectives library can be used in any UPC code by including either its corresponding source file or the whole library. Several algorithms are

implemented per collective operation, relying either on the default algorithm or on the selection by the application programmer/user.

3.2. Implementation of Extended Collectives

According to the structure of the framework presented in the previous section, a description of each group of collective functions is next given, presenting the structure of its arguments and its processing, alongside with examples of use. The algorithms and data movements presented in this section show the basic communications required to implement the corresponding collective operations, which focus on minimizing the amount of data transfers, as well as their size. Besides this, some common optimizations for all these functions implemented according to the execution environment will be commented in Section 3.3. For further information about the interfaces of these functions, the reader is referred to Appendix A.

3.2.1. In-place Collectives

The in-place collectives use only one array as the source and destination of the data involved in the operation. Listing 3.1 presents the signatures of two representative in-place collectives (broadcast and reduce). As shown here, only one argument is used to specify the source/destination array, in order to provide a simpler syntax. The rest of parameters of these functions are the same as for their standard counterparts.

```
void upc_all_broadcast_in_place (
        shared void *srcdst, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_reduceD_in_place (
        shared void *srcdst, upc_op_t op, size_t nelems,
        size_t blk_size, double (*func)(double,double),
        upc_flag_t sync_mode
);
```

List. 3.1: Signatures of representative in-place collectives

The data movements required to execute these collectives are highly dependent on the operation performed. For instance, an in-place broadcast presents a straightforward implementation, because in this case the source data is moved directly to different locations in remote threads' memory, without any potential risk of data overwriting. This situation is analogous for scatter and gather. However, other implemented collectives (e.g., permute, exchange, allgather) have to operate on the source data locations, therefore it is necessary to implement different levels of synchronization to perform correctly the collective operation.

Regarding the permute collective, its execution can potentially involve the overwriting of all the data from all threads, therefore its algorithm has been implemented using an auxiliary private array in each thread to perform the data exchanges between them, in order to perform a backup of the local data in each thread. The reduce, prefix reduce and allreduce collectives also present a behavior similar to their standard counterparts: first, the data stored on each thread's memory is reduced locally to produce a partial result per thread; after that, all threads are synchronized, and finally the partial results are gathered by one thread (e.g., reduce) or by all threads (e.g., allreduce) to produce the final result. Additionally, in the case of the prefix reduce collective, the results are generated according to the established processing order, thus involving synchronizations to send the partial results from all threads. Regarding the allgather and exchange in-place algorithms, some optimizations have been introduced to minimize the number of communications, thus favoring a more efficient processing.

Figure 3.2 presents the data movements implemented for the in-place allgather collective using 3 threads. Here the numbering at each arrow indicates the order in which each communication is performed, thus the arrows with the same numbering represent parallel data movements. First, each thread moves its source data chunk to its corresponding final location within the shared memory space. Then, a synchronization is needed to make sure that all threads have performed this first copy, otherwise source data could be overwritten. Finally, each thread sends its corresponding chunk to the rest of threads without further synchronizations.

The in-place exchange algorithm is presented in Figure 3.3 using four threads. It uses a concatenation-like procedure [5], including additional logic that avoids the overwriting of source data and also balances the workload among threads. More-

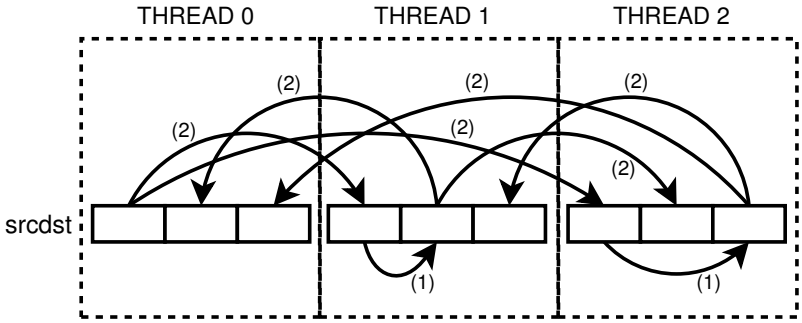


Fig. 3.2: Communications for `upc_all_gather_all_in_place` (3 threads)

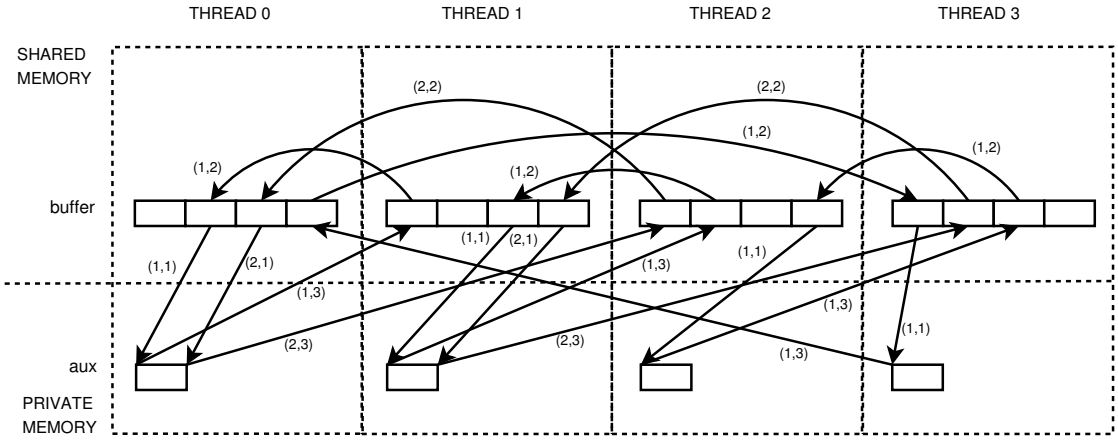


Fig. 3.3: Communications for `upc_all_exchange_in_place` (4 threads)

over, it only needs a single private array of `nbytes` (being this the value passed as parameter to the collective) as extra memory space. This algorithm is performed, at most, in $THREADS/2$ stages. Each stage always consists of three steps: (1) a piece of local source data is moved from shared memory to an auxiliary private array, (2) the corresponding remote data is copied to that source location, and (3) the private memory copy of the source data is moved to the remote location used in the previous step. In the first stage, each thread copies data from/to its right neighbor (i.e., thread i and thread $(i+1)\%THREADS$ are associated), and in the next stages data exchanges continue with the following neighbors (for thread i, it would be thread $(i+s)\%THREADS$, where s is the number of stage). In order to avoid data dependencies, all threads are synchronized after the execution of each stage. When the number of threads is even, the last stage only needs to be performed by half of the threads (in this implementation, the threads with an identifier less than

THREADS/2). The arrow numbering of Figure 3.3 consists of two values, that indicate the number of stage (left) and step (right) in which the data movement is performed. No synchronizations are required between steps in the same stage, and in the ideal scenario all communications with the same numbering would be executed in parallel.

Additionally, a rooted version has been implemented for the four in-place collectives where it is possible to define a root thread (broadcast, scatter, gather and reduce). Listing 3.2 presents the signature of the rooted in-place broadcast collective as an example of them. As commented before, these functions take the same arguments as the corresponding in-place collectives, but including the identifier of the root thread (parameter `root`). Their internal implementation is also very similar to their associated extended function, but changing the source (broadcast, scatter) or the destination (gather, reduce) address according to the given root thread.

```
void upc_all_broadcast_rooted_in_place (
        shared void *srcdst, size_t nbytes, int root,
        upc_flag_t sync_mode
);
```

List. 3.2: Signature of a representative rooted in-place collective

The main advantage of in-place collectives is that they operate on a single array without requiring the allocation of auxiliary arrays to implement data movements, as this additional space is managed transparently to the user by the collective. To illustrate their use, a common routine for time measuring is presented in Listing 3.3. The use of the selected extended collective (`upc_all_reduceD_all_in_place`) returns the final result in UPC shared memory, accessible by all threads, using only a shared array of *THREADS* elements (`times`).

```
shared double times[THREADS];
...
times[MYTHREAD] -= getCurrentTime();
...
times[MYTHREAD] += getCurrentTime();
upc_all_reduceD_all_in_place(times, UPC_MAX, THREADS, 1, NULL,
    sync_mode);
```

List. 3.3: Time measuring routine using in-place collectives

3.2.2. Vector-variant Collectives

The vector-variant collectives allow the definition of a variant number of elements communicating at each thread. The extended collectives library includes vector-variant implementations for the eight standard UPC collectives plus allreduce, but it additionally includes a general memory copy function named `upc_all_vector_copy`. This function performs a custom number of data movements between any pair of threads, allowing the user to specify the displacement and number of elements for each communication, thus supporting a high level of flexibility in the library. The signatures of five representative vector-variant collectives are included in Listing 3.4.

```
void upc_all_broadcast_v (
  shared void *dst, shared const void *src, shared int *ddisp,
  size_t nelems, size_t dst_blk, size_t typesize,
  upc_flag_t sync_mode
);
void upc_all_gather_v (
  shared void *dst, shared const void *src, shared int *sdisp,
  shared size_t *nelems, size_t dst_blk, size_t src_blk,
  size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v (
  shared void *dst, shared const void *src, shared int *ddisp,
  shared int *sdisp, shared size_t *nelems, size_t dst_blk,
  size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_reduceI_v (
  shared void *dst, shared const void *src, upc_op_t op,
  shared int *sdisp, shared size_t *nelems, int nchunks,
  size_t src_blk, int (*func)(int,int), upc_flag_t sync_mode
);
void upc_all_vector_copy (
  shared void *dst, shared const void *src, shared int *ddisp,
  shared int *sdisp, shared size_t *nelems, int nchunks,
  size_t dst_blk, size_t src_blk, size_t typesize,
  upc_flag_t sync_mode
);
```

List. 3.4: Signatures of representative vector-variant collectives

These extended collectives present two main distinctive arguments: the number of elements that are moved by each thread (`nelems`) and the size of the data type of the elements in the arrays (`typesize`). The use of these two arguments is intended to provide an intuitive interface for the collective, because the user does not need to deal directly with data sizes in communication. Besides `nelems` and `typesize`, the vector-variant collectives also include a variable number of arguments that are used to define their associated data transfers, according to their type of processing. For example, the block size of an array is necessary in order to perform the exact indexing of a particular array position in terms of `nelems` and `typesize` following the UPC pointer arithmetic. As a result, the vector-variant broadcast and scatter require the definition of the block size of the destination array (`dst_blk` in Listing 3.4), whereas gather needs the value of the block size of the source array (`src_blk` in Listing 3.4). Additionally, other parameters related to displacements on source/destination arrays (`sdisp` and `ddisp`, respectively) are also used by vector-variant collectives to define an offset from the given starting position of the associated array. These arguments are associated to the previous definitions of block sizes, i.e. if one of these collectives requires the definition of the block size parameter for the source/destination array, this function should also include a parameter to define an offset on it. In general, as Listing 3.4 shows, additional array parameters are defined as shared in order to favor a global view of their values, even though the access to these variables is internally privatized for each thread to avoid performance bottlenecks.

Most of the standard data-movement collectives (scatter, gather, allgather, exchange and permute) present vector-variant implementations that follow a similar structure: all the arrays that define the number of elements in each communication (and also the displacements, when present) have *THREADS* elements. The broadcast collective simplifies the interface by using a single shared scalar value as the size of communications. Nevertheless, the implementation of reduce, prefix reduce, allreduce and vector copy differs significantly from the previous ones, because the use of different size of communications per thread should also cover the possibility of having more than one data movement per thread. In fact, the best option in terms of programmability is to let the user define a custom number of chunks in the source array to execute the collective regardless of their thread affinity, thus this approach has been used for these four collectives. Figure 3.4 shows the operation of a call to `upc_all_reduceI_v` using three chunks, whose communications associated

to the reduced data are labeled with the same number as in the arrays containing displacements (`sdisp`) and elements per chunk (`nelems`).

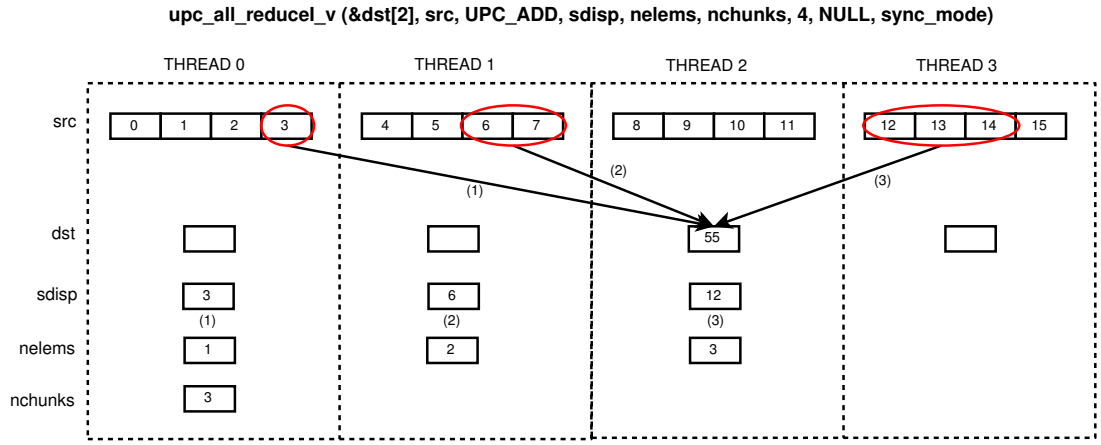


Fig. 3.4: Communications for `upc_all_reduceI_v` (4 threads)

In addition to this, the extended library implements an optimized operation when all threads define the same number of chunks. This algorithm is activated by multiplying the number of chunks by the predefined constant `UPC_EXTCOLLS_VV_CHUNKS`, and the only requirement is that the description of each chunk has to be specified thread by thread following a cyclic order in the corresponding arguments (i.e., `sdisp`, `ndisp`, `nelems`): the element i in these arguments must correspond to a chunk associated to thread $i\%THREADS$. Listing 3.5 presents the definition of the function call presented in Figure 3.4, but adding a chunk for thread 2 (thus processing one element in position 9 of array `src`) in order to activate the optimized algorithm.

```
sdisp = {3, 6, 9, 12};
nelems = {1, 2, 1, 1};
nchunks = UPC_EXTCOLLS_VV_CHUNKS * 4;
upc_all_reduceI_v(&dst[2], src, UPC_ADD, sdisp, nelems, nchunks,
                  4, NULL, sync_mode);
// The new result in dst[2] is 64
```

List. 3.5: Activation of optimized processing for `upc_all_reduceI_v`

Four additional versions of these functions have been included in the library (the corresponding signatures for the vector-variant broadcast and allgather are shown in Listing 3.6):

```
void upc_all_broadcast_v_rooted (
  shared void *dst, shared const void *src, shared int *ddisp,
  size_t nelems, size_t dst_blk, size_t typesize, int root,
  upc_flag_t sync_mode
);
void upc_all_broadcast_v_raw (
  shared void *dst, shared const void *src, shared size_t *ddisp_raw,
  size_t nbytes, size_t dst_blk_raw, upc_flag_t sync_mode
);
void upc_all_broadcast_v_local (
  shared void *dst, shared const void *src, shared int *ddisp_local,
  size_t nelems, size_t dst_blk, size_t typesize,
  upc_flag_t sync_mode
);
void upc_all_gather_all_v_raw (
  shared void *dst, shared const void *src, shared int *ddisp_raw,
  shared int *sdisp_raw, shared size_t *nbytes, size_t dst_blk_raw,
  size_t src_blk_raw, upc_flag_t sync_mode
);
void upc_all_gather_all_v_local (
  shared void *dst, shared const void *src, shared int *ddisp_local,
  shared int *sdisp_local, shared size_t *nelems, size_t dst_blk,
  size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_privparam (
  shared void *dst, shared const void *src, int *ddisp,
  int *sdisp_local, size_t *nelems, size_t dst_blk, size_t src_blk,
  size_t typesize, upc_flag_t sync_mode
);
```

List. 3.6: Signatures of representative versions of vector-variant collectives

- *rooted*: these versions (only available for broadcast, scatter and gather) include the label `_rooted` and an additional integer argument (the root thread).
- *raw*: these versions (labeled with `_raw` and available for all collectives except for reduce and prefix reduce) allow the user to define the number of bytes transferred by each thread analogously to the standard collectives, instead of using the number of elements and the element size. Therefore, they use a parameter `shared size_t *nbytes` instead of parameters `nelems` and `typesize`.

- *local*: these functions (with the label `_local` at the end of their names) provide the possibility of defining array displacements as relative positions inside a thread, instead of using the default absolute array values. They are available for broadcast, scatter, gather, allgather, exchange and permute, because the extended function must perform only one communication per thread: the reason is that each thread is assigned a value in arrays of *THREADS* elements (e.g., `sdisp`, `ddisp`) whose index is equal to the identifier of the given thread.

  In order to illustrate their behavior, Figure 3.5 presents three function calls that perform the same data movements, but using different versions of the vector-variant broadcast with the consequent changes in their arguments.

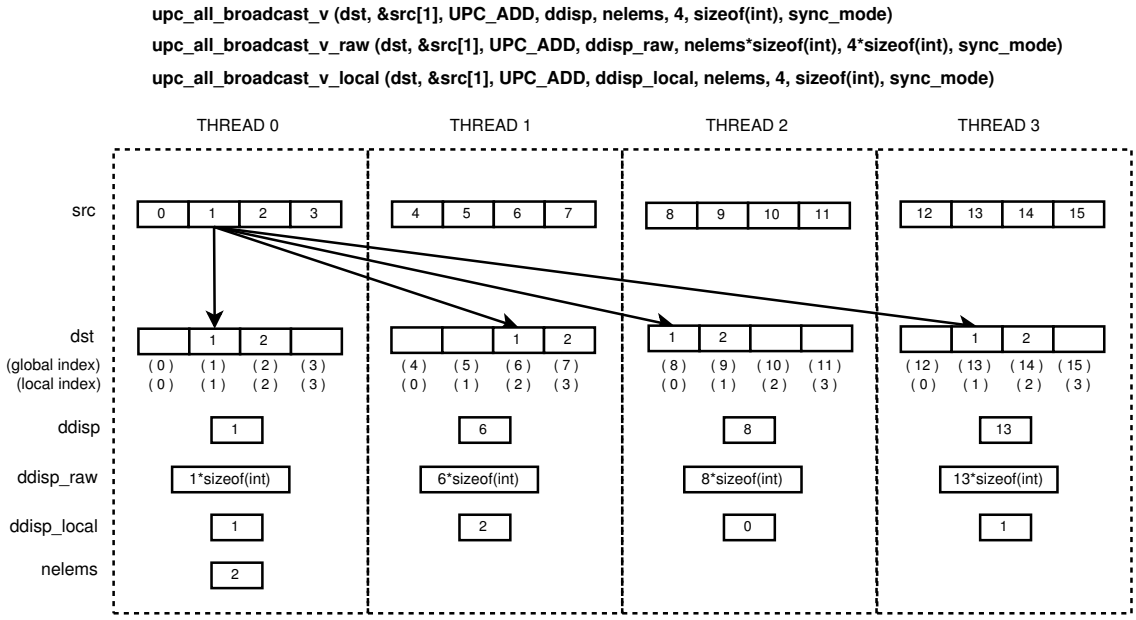


Fig. 3.5: Communications for different versions of `upc_all_broadcast_v` (4 threads)

- *privparam*: these versions (only available for allgather, exchange and vector copy) take the parameters of source/destination displacements and number of elements as private variables. As these three functions perform multiple accesses to these arrays in order to obtain the source and destination locations for each data chunk, performance and scalability can be improved by keeping these data in private memory.

Additionally, a *merge* version for the vector-variant exchange has been implemented. The difference lies in the way the elements are gathered by each thread: the `upc_all_exchange_v` collective copies each chunk to the same relative position in the destination array as in the source thread, whereas the `_merge` version puts all chunks in consecutive memory locations. It uses an additional array argument with *THREADS* positions that indicates the location of the first element copied from thread 0 to each thread, and the rest of the elements are copied consecutively using that value as reference.

An example of use of the vector-variant collectives is the copy of an upper triangular matrix from vector `A` to `B`, which is implemented in Listing 3.7. Here the initialization consists in setting the displacement arrays for the source and destination addresses, as well as the number of elements for each of the N rows (chunks) of the matrices. After that, a call to `upc_all_vector_copy` is enough to perform all necessary data movements.

```
shared [N*N/THREADS] int A[N*N], B[N*N];
shared int sdisp[N], ddisp[N], nelems[N];
// Initialization of shared argument arrays
upc_forall (i=0; i<N; i++; &sdisp[i]) {
    sdisp[i]=i*N+i; ddisp[i]=i*N+i;
    nelems[i]=N-i;
}
upc_barrier;
upc_all_vector_copy(B, A, ddisp, sdisp, nelems, N,
    N*N/THREADS, N*N/THREADS, sizeof(int), sync_mode);
```

List. 3.7: Copy of a triangular matrix using vector-variant collectives

3.2.3. Team-based Collectives

These collectives are based on teams, which are subsets of the UPC threads running an application. The use of teams has been addressed by the UPC community, mainly focusing on an implementation at the language level [57, 70], although the use of MPI has also been suggested [16]. However, up to now no standard UPC team implementation has been defined. In order to overcome this limitation and support the use of teams in collectives, this subsection presents a library-based support for

UPC teams, which uses a structure to define the necessary variables to implement them. After its introduction, the developed team-based collectives are described.

Library Support for Teams

Listing 3.8 presents the struct data type that defines a team. It uses an array of *THREADS* boolean elements (`isThreadInTeam`) that indicate whether a thread is included in the team or not. A team identifier (tid) for each thread is assigned in increasing order of the UPC thread identifier. The variable `numthreads` indicates the number of threads in the team. The `counterBarrier` and `flagBarrier` arrays include two thread counters and two flags, respectively, that are used as auxiliary variables for the implementation of synchronization barriers through an active-wait algorithm. The lock variable `lockTeam` is used to implement atomic operations in the team (e.g., in the execution of team barriers or team management operations, such as thread insertions). Finally, the `pointerArg` variable is an auxiliary array used for memory allocation within the team. Using these variables, this implementation is able to provide full support for team-based collectives.

```
struct teamContent {
  shared t_boolean *isThreadInTeam;   // THREADS elements
  shared int *numthreads;
  shared int *counterBarrier;         // 2 elements
  shared int *flagBarrier;            // 2 elements
  upc_lock_t *lockTeam;
  shared void *shared *pointerArg;    // THREADS elements
};
typedef struct teamContent team;
```

List. 3.8: Structure for UPC teams support

Listing 3.9 presents the signatures of the auxiliary functions that have been implemented to support teams handling. The main functions included in this interface are the insertion of a thread in a team, tests for inclusion and a synchronization barrier. Apart from these basic functionalities, memory allocation functions have also been included to support the team management, because some of the standard UPC routines for memory allocation are collective (namely `upc_all_alloc` and `upc_all_lock_alloc`), and therefore cannot be used by a subset of threads.

```
// Initialize team
void init(team *t);
// Get number of threads in a team
int numThreads(team t);
// Insert thread in team
void insert(int thread, team t);
// Implementation of a barrier
void teamBarrier(team t);
// Test if the thread with the given identifier
// is included in a team
int isThreadInTeam(int threadID, team t);
// Get tid for a given thread ID
int getTeamID(int threadID, team t);
// Get thread ID for a given tid
int getThreadID(int teamID, team t);
// Polymorphic function to allocate shared team variables
shared void *allocate(int numblocks, int blksize, team t);
// Polymorphic function to allocate team locks
upc_lock_t *allocateLock(team t);
// Deallocate shared variables defined using "allocate"
void deallocate(shared void *pointer, team t);
// Deallocate locks defined using "allocateLock"
void deallocateLock(upc_lock_t *pointer, team t);
```

List. 3.9: Signatures of the team management auxiliary functions

It is important to note that the team-based collectives included in the developed library do not depend on a specific definition of a team, because team manipulation is always performed through calls to the functions in Listing 3.9. Therefore, any underlying implementation of a team and its auxiliary functions can be used to call the collectives, and the proposed signatures only represent a possible naming convention for a team management interface. The team-based collectives have been implemented independently from the underlying team library, by dealing with teams through different management functions for basic team operations, such as barriers or memory allocation routines. Using these functions as interface, the separation between collectives and team libraries is achieved.

Description of Team-based Collectives

All team-based collectives (labeled with `_team`) present the same arguments as their standard counterparts, plus the private variable that represents the team de-

scription. Listing 3.10 shows the signature of the team-based gather collective as a representative example. The team-based implementation interprets the argument that represents the size of communications (`nbytes`) as the total amount of data that is transferred by all threads in the team. Thus, `nbytes/numthreads` bytes are transferred by each thread, and the first chunk goes to the thread with tid 0 (being "tid" the identifier within a team, which may not be necessarily equal to the associated thread's identifier). Only the members of the team can invoke these functions.

```
void upc_all_gather_team (
   shared void *dst, shared const void *src, size_t nbytes,
   team t, upc_flag_t sync_mode
);
```

List. 3.10: Signature of a team-based collective

Additionally, the scatter, gather, allgather and exchange collectives admit the implementation of a filter version (labeled with `_team_allthr`), in which the team only prevents the threads that are not included in it from executing the operation. Therefore, this version ignores team identifiers for communication, and the argument `nbytes` is interpreted as the amount of data transferred by each thread. All these filter operations have the same argument types as the `_team` counterpart.

Figure 3.6 illustrates the behavior of both types of team-based collectives with a scatter collective executed using 4 threads. Both functions have a source array of 12 KB, which is distributed according to the definition of each collective among the three threads of team `t` (0, 1 and 3, that have tids 0, 1 and 2, respectively).

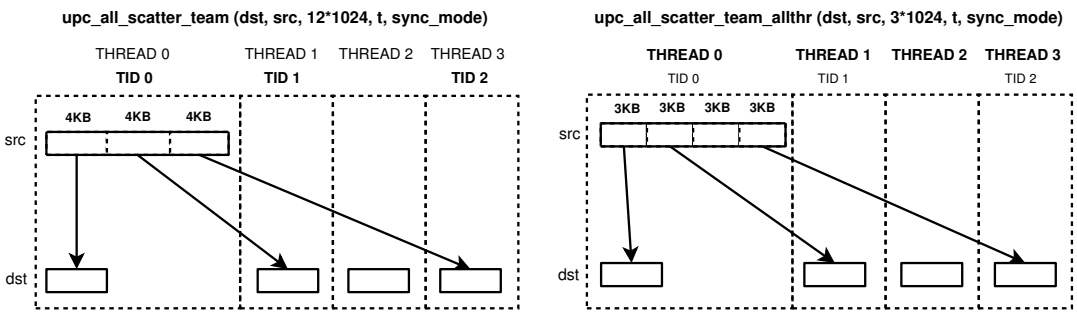


Fig. 3.6: Communications for team-based scatter operations (4 threads)

Finally, the piece of code in Listing 3.11 presents an example of the use of team-based collectives: two teams execute the same function (computation of Pi using the Monte Carlo method) in parallel in the same UPC program. This allows the exploitation of heterogeneous architectures, supporting resources with different computational power: a team could group different processors (even hardware accelerators such as GPUs) according to their features, thus helping handle workload imbalance.

```
void computePiMontecarlo(int trials, team t,
  shared double *estimation) {
    shared int global_hits[THREADS];
    shared int local_hits[THREADS];
    double piEstimation;
    // Filter all threads that are not included in the team
    if (!isThreadInTeam(MYTHREAD,t)) return;
    // Auxiliary function
    int nth = getNumThreads(t);
    // Compute local hits, put result in local_hits[MYTHREAD]
    ...
    upc_all_reduceI_all_team(global_hits, local_hits, UPC_ADD,
      nth, 1, NULL, t, sync_mode);
    // Compute pi estimation
    ...
    *estimation = piEstimation;
    return;
}

int main() {
    team t1, t2;
    shared double est1, est2;
    // Initialize variables and create teams (disjoint sets)
    ...
    // Execute tasks
    computePiMontecarlo(trials1, t1, &est1);
    computePiMontecarlo(trials2, t2, &est2);
    if (MYTHREAD == 0) {
      printf("Estimation:%lf\n",
         (est1*trials1+est2*trials2)/(trials1+trials2));
    }
}
```

List. 3.11: Computation of Pi using team-based collectives

3.2.4. Get-put-priv Collectives

The get-put-priv collectives allow the use of a private array as source and/or destination parameter in all the previously discussed extended collectives (in-place, vector-variant and team-based, alongside their own additional versions), and also in the standard collectives plus allreduce. They represent the largest subset of collectives included in the extended library, as the commented functionality can be applied to the standard UPC collectives and also to the rest of extended collectives. The get-put-priv functions are classified in three subgroups:

- *get collectives*: shared source and private destination.
- *put collectives*: private source and shared destination.
- *priv collectives*: the source and destination are both private.

In-place collectives are the only subset that cannot implement all these three variants, but only the *priv* one, as they have the same array as source and destination. The only change introduced in the arguments of these functions compared to standard/extended collectives is the declaration of the source and/or destination array(s) as private. For illustrative purposes, Listing 3.12 shows the signatures of the allgather get-put-priv versions, both standard and vector-variant.

The algorithms implemented in these collectives minimize the number of communications, avoiding unnecessary remote data transfers and maximizing parallel processing among threads using as few synchronization points as possible. In general, *get* and *put* collectives do not use any additional buffer in private or shared memory to implement the necessary data copies: each thread uses its associated shared memory space in the source (*get*) or the destination (*put*) for all communications. Only *priv* collectives require the allocation of additional shared memory to allow the data transfers, at most the same size as the communications performed. As an example, Figure 3.7 shows the *priv* version of a broadcast that uses 4 threads. First, thread 0 stores its data in an auxiliary buffer in shared memory, then a synchronization is necessary to make sure that the data has been made available, and finally all threads copy the data to the final location. Here thread 0 copies its own data in its private memory space using the `memcpy` system library routine.

```
void upc_all_gather_all_get (
  void *dst, shared const void *src, size_t nbytes,
  upc_flag_t sync_mode
);
void upc_all_gather_all_put (
  shared void *dst, const void *src, size_t nbytes,
  upc_flag_t sync_mode
);
void upc_all_gather_all_priv (
  void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_all_v_get (
  void *dst, shared const void *src, shared int *ddisp,
  shared int *sdisp, shared size_t *nelems, size_t dst_blk,
  size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_put (
  shared void *dst, const void *src, shared int *ddisp,
  shared int *sdisp, shared size_t *nelems, size_t dst_blk,
  size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_priv (
  void *dst, const void *src, shared int *ddisp,
  shared int *sdisp, shared size_t *nelems, size_t dst_blk,
  size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
```

List. 3.12: Signatures of representative get-put-priv collectives

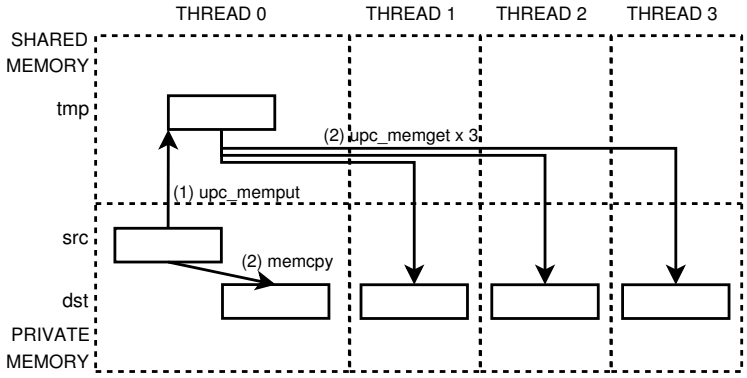


Fig. 3.7: Data movements for `upc_all_broadcast_priv` (4 threads)

The usefulness of these functions can be assessed in a parallel image filtering algorithm, presented in Listing 3.13. The input data, a matrix of $N \times N$ elements, is stored in the private memory of thread 0 (in matrix `img`), and then the private versions of the standard scatter and in-place broadcast perform the necessary data movements to distribute the workload to the private memory spaces of all threads (in matrix `aux`). The filtering algorithm is executed on private memory to favor the efficient exploitation of data locality, because computing with private memory is more efficient than dealing with shared memory [22]. Finally, the private version of the standard gather collective returns the final result in the private memory of thread 0 (in matrix `filteredImg`).

```
// Initialize 'img' as the source N*N image, 'aux' as an
//  auxiliary array of (N/THREADS)*N elements on each thread
//  and 'filter' as a 3x3  matrix.
// All private variables are initialized on thread 0

upc_all_scatter_priv(aux, img, (N/THREADS)*N*sizeof(double),
  sync_mode);
upc_all_broadcast_in_place_priv(filter, 3*3*sizeof(double),
  sync_mode);

filterMatrix(aux, filter, N, N, 3, 3);

upc_all_gather_priv(filteredImg, aux, (N/THREADS)*N*sizeof(double),
  sync_mode);
```

List. 3.13: Image filtering algorithm using get-put-priv collectives

3.3. Optimization of Extended Collectives

The functions included in the extended collectives library have been implemented efficiently, in order to take advantage of the underlying system architecture. As the main goal of their development is to provide high programmability and flexibility, any improvement seeking better performance for these collectives has avoided the introduction of restrictions in their applicability, i.e. no platform-specific or implementation-specific code has been used. Therefore, the basis of these optimizations are always standard UPC and C functions, which have been used to implement efficiently the collective algorithms.

In general, the implemented optimizations focus on obtaining more efficient algorithms for hybrid shared/distributed memory communications, in order to minimize internode communications, mainly managing efficiently remote data transfers. The use of straightforward communications in a minimum number of steps, as shown for the algorithms in the previous section (e.g., the maximization of parallel operations in Figure 3.7), is only reserved for pure shared memory communications: these transfers benefit from the high memory capabilities of multicore systems, which can handle efficiently multiple large-sized parallel memory copies (see Section 2.1.2 for more information). However, the interconnection networks present, in general, a high start-up latency and have a limited bandwidth, demanding generally tree-based algorithms to handle efficiently data movements and auxiliary memory storage. In the extended library, three different tree implementations have been used.

Figure 3.8 shows a representative flat-tree algorithm for multicore systems using `upc_all_scatter_priv`, that illustrates the remote and local data movements performed on 4 nodes with 4 threads on each node using 16 KB of source data. This algorithm has been implemented to minimize remote data transfers on clusters of multicore nodes, but also maximizing parallel communications. Here, thread 0 divides the source data in equal parts for each node, and sends each part to a different node, receiving the data a thread on behalf of all the threads of the node. After that, the thread that receives the data performs an intranode scatter (in shared memory) of the received 4 KB data chunks to the rest of threads in the node. The label "(i)" marks the execution step in which the associated data movement is performed.

Figure 3.9 presents an alternative implementation of `upc_all_scatter_priv` using a binomial tree, which is focused on NUMA node architectures. In this algorithm, thread 0 begins the scatter of the first half of the total source data to nodes 0 and 1 in the first step, similarly to the previous flat-tree algorithm, and also sends the remaining source data to the root in node 2 (thread 8), which in turn starts scattering these data to nodes 2 and 3 in the following step. It is also important to note that a thread performing remote and private data movements in the same execution step (e.g., thread 0 in step 1) will prioritize the remote ones in order to favor the throughput of the algorithm. According to Figure 3.9, this algorithm is implemented in more time steps than the flat-tree one for multicore systems, but it provides a more scalable implementation because of

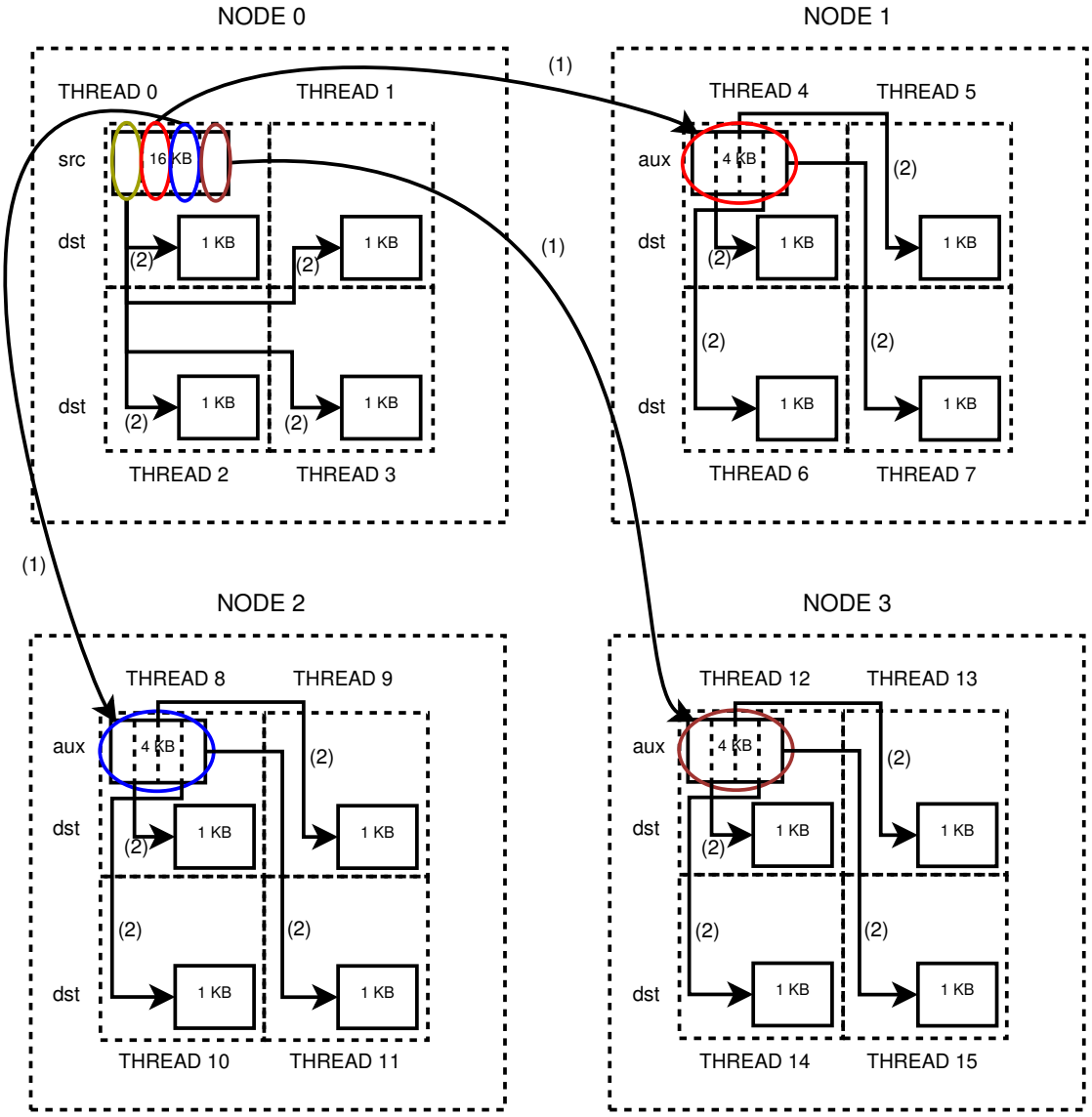


Fig. 3.8: Data movements for `upc_all_scatter_priv` with flat-tree algorithm on 4 nodes

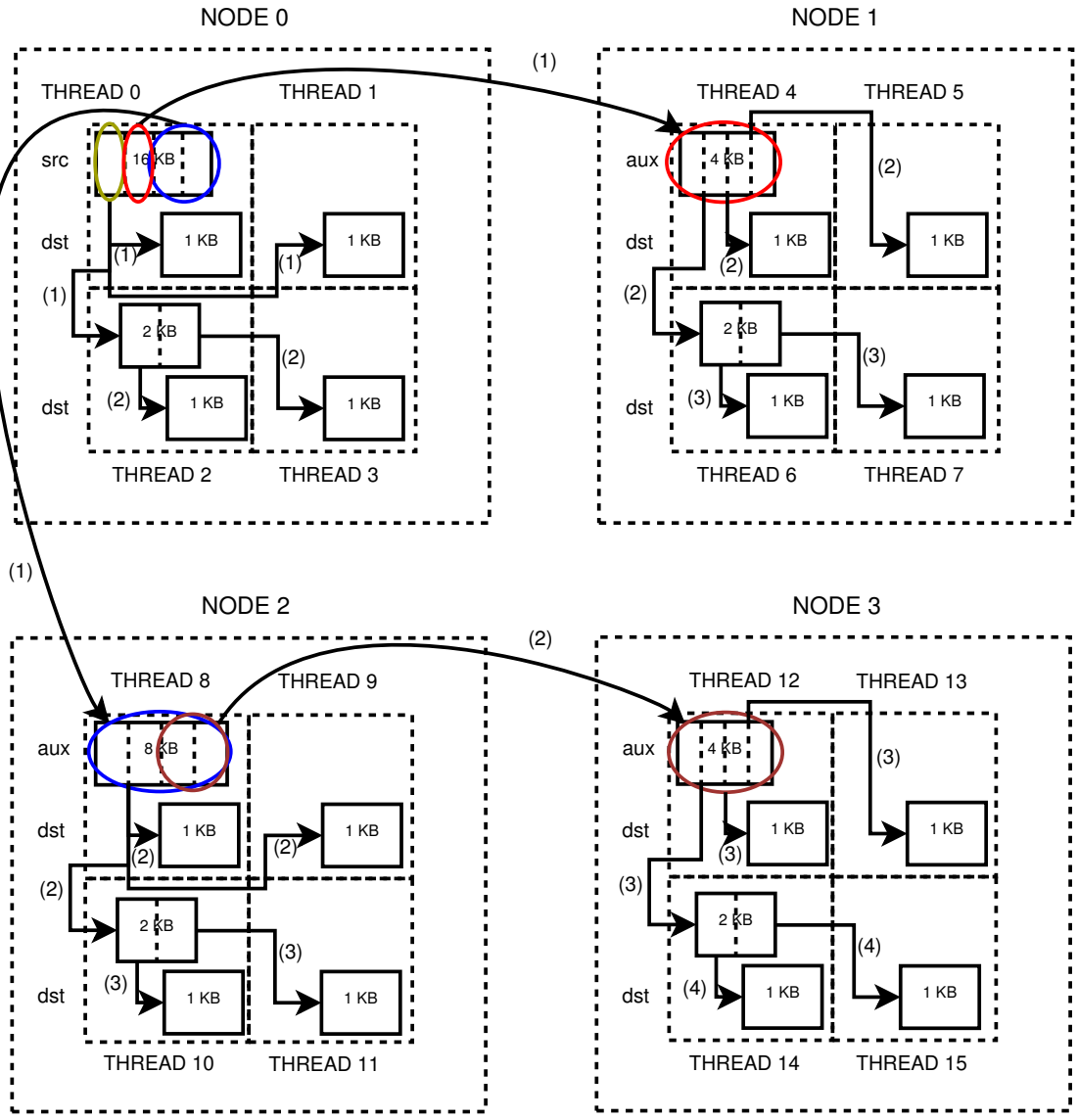


Fig. 3.9: Data movements for `upc_all_scatter_priv` with binomial-tree algorithm on 4 nodes

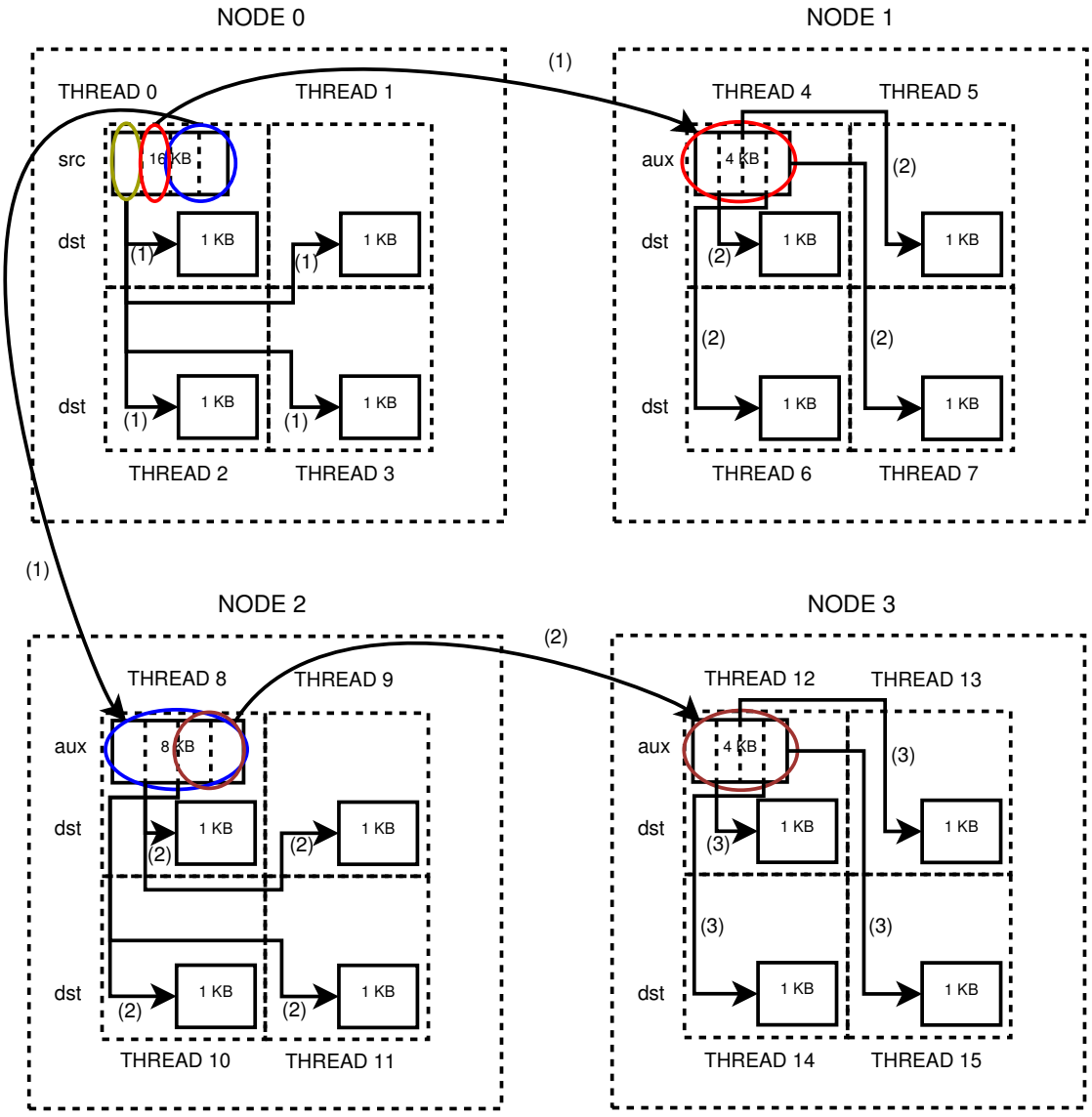


Fig. 3.10: Data movements for `upc_all_scatter_priv` with binomial-flat-tree algorithm on 4 nodes

the stepwise remote accesses to the source data on thread 0: when the number of nodes is large, the flat-tree approach presents too many concurrent accesses to the source data, and the binomial tree splits these accesses between different nodes, thus favoring the use of different data paths.

A hybrid communication pattern between the two previous algorithms, called binomial-flat-tree approach, is illustrated in Figure 3.10. Here all internode communications are performed following a binomial-tree approach, whereas all intranode communications follow the flat-tree algorithm. This approach is useful on multicore nodes with a UMA-like architecture, thus helping to exploit memory bandwidth more efficiently and reducing synchronizations between threads.

In order to complement these algorithms, additional optimizations related to memory handling and thread affinity have been implemented in the library. The most relevant ones are: (1) the segmentation of large-sized internode communications, (2) the reutilization of shared auxiliary buffers for different collective functions, and (3) the use of thread pinning in order to assign the processing of a given thread to a selected core in the system. All these features are configurable by the library, and can be activated on demand depending on the execution requirements.

3.4. Use of Extended Collectives: Case Studies

The extended collectives improve programmability for a wide variety of problems by reducing the number of SLOCs and favoring code expressiveness. Nevertheless, their adoption depends on the achievement of relevant benefits when compared to their equivalent implementation in standard UPC. Thus, this section analyzes the impact of extended collectives on different codes (dense and sparse matrix multiplication, Integer Sort and 3D Fast Fourier Transform), justifying the benefits in terms of programmability obtained from their use.

3.4.1. Dense Matrix Multiplication Kernel

Listing 3.14 presents an optimized standard UPC code that multiplies two dense `N`×`N` matrices (`C`=`A`×`B`). The source matrices `A` and `B` are stored in the private mem-

```
#define chunk_size N*N/THREADS;
double *A, B[N*N], *C;
double local_A[chunk_size], local_C[chunk_size];
shared [chunk_size] double temp_A[N*N], temp_C[N*N];
shared [] double temp_B[N*N];
if (MYTHREAD == 0) {
  // Allocate and initialize arrays A and C, initialize B
  ...
  memcpy(local_A, A, chunk_size*sizeof(double));
  for (i=1; i<THREADS; i++) {
     upc_memput(&temp_A[i*chunk_size], &A[i*chunk_size],
        chunk_size*sizeof(double));
  }
}
upc_barrier;
if (MYTHREAD != 0) {
   upc_memget(local_A, &temp_A[MYTHREAD*chunk_size],
      chunk_size*sizeof(double));
}
if (MYTHREAD == 0) {
   upc_memput(temp_B, B, N*N*sizeof(double));
}
upc_barrier;
if (MYTHREAD != 0) {
   upc_memget(B, temp_B, N*N*sizeof(double));
}

computeSubmatrix(local_A, B, local_C, N/THREADS, N, N);

if (MYTHREAD != 0) {
   upc_memput(&temp_C[MYTHREAD*chunk_size], local_C,
      chunk_size*sizeof(double));
}
upc_barrier;
if (MYTHREAD == 0) {
   memcpy(C, local_C, chunk_size*sizeof(double));
   for (i=1; i<THREADS; i++) {
      upc_memget(&C[i*chunk_size], &temp_C[i*chunk_size],
         chunk_size*sizeof(double));
    }
}
```

List. 3.14: Original UPC dense matrix multiplication code

ory of thread 0. All matrices are stored in a linearized form (1D) according to the UPC standard. To parallelize the operation, matrix `A` is split in chunks, distributed evenly among all threads and stored in their private memory spaces together with a copy of matrix `B`, which is broadcast from thread 0. After the local multiplication, the result matrix is finally gathered in thread 0. All the data movements between threads are performed using multiple one-sided communications with memory copy functions, because of the lack of collectives support for operating with different private memory spaces as source and destination addresses, and thus auxiliary shared arrays (`temp_A`, `temp_B` and `temp_C`) and synchronizations (three calls to `upc_barrier`) are necessary to perform the data transfers between threads.

The use of extended collective functions can reduce significantly the complexity of the UPC implementation of this kernel, as presented in Listing 3.15. Here all the data movements associated to the source and destination arrays are implemented using extended collective functions with private arguments. Matrix `A` is evenly distributed to all threads using a *priv* scatter collective, whereas a *priv* in-place broadcast transfers the whole matrix `B` to all threads. Finally, the result matrix `C` is obtained using a *priv* gather collective. It is important to note that the programmer does not need to deal with any temporary buffer to perform the communications, as the extended collectives handle the auxiliary memory space transparently to the user. Therefore, extended collectives allow the parallelization of this code without requiring the user to deal with shared memory addresses or temporary buffers, and additionally the user takes advantage of efficient communication algorithms for data transfers transparently.

```
double local_A[chunk_size], local_C[chunk_size];
upc_all_scatter_priv(local_A, A, (N*N/THREADS)*sizeof(double),
    sync_mode);
upc_all_broadcast_in_place_priv(B, N*N*sizeof(double),
    sync_mode);

computeSubmatrix(local_A, B, local_C, N*N/THREADS, N, N);

upc_all_gather_priv(C, local_C, (N*N/THREADS)*sizeof(double),
    sync_mode);
```

List. 3.15: UPC dense matrix multiplication code with extended collectives

3.4.2. Sparse Matrix Multiplication Kernel

This kernel performs the multiplication of a sparse matrix, stored in Compressed Sparse Row (CSR) format, by a dense matrix. Here the work distribution is done by subdividing the different arrays that define the compressed sparse matrix (values, column index and row pointer), selecting the number of rows that each thread should process to obtain balanced workloads. As the number of elements in each array can be different for each thread, the scatter and gather operations are performed using vector-variant collectives.

```
// Initialize variables: 'val', 'col_ind', 'row_ptr' (values,
//  column index and row pointer arrays in CSR format), 'B'
//  ('k'*'n' dense source matrix), 'C', 'C_dist' (final
//  'm'*'n' and partial result matrices), 'nz' (number of
//  non-zero values), 'disp...' (some displacement vectors),
//  'nelems', 'nrows' (element size parameters)
...
upc_all_scatter_v_priv(val_dist, val, disp, nelems, nz,
  sizeof(double), sync_mode);

upc_all_scatter_v_priv(col_ind_dist, col_ind, disp, nelems, nz,
  sizeof(int), sync_mode);

upc_all_vector_copy_priv(row_ptr_dist, row_ptr, disp_rows_dst,
  disp_rows_src, nrows, THREADS, m+1, m+1, sizeof(int), sync_mode);

upc_all_broadcast_in_place_priv(B,  k*n*sizeof(double), sync_mode);

// Modify 'nrows' to allow separate calls to the multiplication
//  algorithm on each thread
...
computeMMSparse(val_dist, col_ind_dist, row_ptr_dist, B, C_dist,
  nrows[MYTHREAD], k, n);

// Modify variables to gather the computed submatrices
...
upc_all_gather_v_priv(C, C_dist, disp, numvalues, m*n,
  sizeof(double), sync_mode);
```

List. 3.16: UPC sparse matrix multiplication code with extended collectives

Listing 3.16 shows the most relevant parts of the sparse matrix multiplication code that use extended collectives. The multiplication is performed by calling the sequential multiplication routine separately on each thread, and therefore this in-

volves some small modifications to the CSR arrays (both for standard UPC and using the extended library): each sparse matrix chunk is processed by the corresponding thread as an independent matrix. Once the results of each submatrix are calculated, the displacements are updated accordingly to gather all chunks correctly in the private memory of thread 0.

The equivalent standard UPC implementation requires a significantly higher number of SLOCs in order to support the vector-variant data transfers using loops and array subscripts. Therefore, this code is not shown for clarity purposes.

3.4.3. Integer Sort Kernel

The Integer Sort (IS) kernel [56] from the NAS Parallel Benchmark (NPB) suite for UPC [32] has been traditionally used in UPC benchmarking [20, 46, 104]. The core of the kernel is the `rank` function, which performs the bucket sort of a set of integer keys, and a piece of its code consists in redistributing the keys by means of an all-to-all operation with data chunks of different sizes.

Listing 3.17 presents the original implementation of the data exchange performed in the `rank` function of IS. The keys are stored in a shared array (`key_buff1_shd`), and the information about the data chunks that correspond to each thread is stored in a private array of *THREADS* structures (`infos`). Each structure in this auxiliary array contains the number of elements and the offset of the first element for each data chunk. The chunks received by a thread after the all-to-all communication are stored consecutively in its private memory (array `key_buff2`).

Listing 3.18 shows the implementation of the all-to-all communications of `rank` in IS using the *get* version of the `upc_all_exchange_v_merge_local` extended collective. As the displacements used are relative array positions, a *local* version is required, and the *get* variant is necessary to use the same source and destination arrays as in the original code. However, the extended collective handles the displacements (`send_displ_shd`) and element counts (`send_count_shd`) separately, thus the array of structs is split in two separate shared arrays. Additionally, a displacement vector (`disp`) is required by the collective call to indicate the offset for the first element received at the destination (set to *0* for all threads in this code) as

```
// Initialization of variables and data types
// ...
upc_barrier;
for (i=0; i<THREADS; i++) {
  upc_memget(&infos[i], &send_infos_shd[MYTHREAD][i],
      sizeof(send_info));
}
for (i=0; i<THREADS; i++) {
  if (i == MYTHREAD)
    memcpy(key_buff2 + total_displ, key_buff1 + infos[i].displ,
        infos[i].count * sizeof(INT_TYPE));
  else
    upc_memget(key_buff2 + total_displ,
        key_buff1_shd+i+infos[i].displ*THREADS,
        infos[i].count * sizeof(INT_TYPE));
  total_displ += infos[i].count;
}
upc_barrier;
```

List. 3.17: Original UPC code in Integer Sort

```
// Initialization of variables and data types
...
upc_all_exchange_v_merge_local_get(
  key_buff2, key_buff1_shd, send_displ_shd, send_count_shd,
  disp, SIZE_OF_BUFFERS, SIZE_OF_BUFFERS, sizeof(int), sync_mode);
```

List. 3.18: UPC code in Integer Sort using extended collectives

well as the block size of the source array (`SIZE_OF_BUFFERS`), which is a predefined constant in the code. For illustrative purposes, a sample code using the function `upc_all_exchange_v_merge_local` in a 2-thread scenario is shown in Figure 3.11.

3.4.4. 3D Fast Fourier Transform Kernel

The 3D Fast Fourier Transform (FFT) is another kernel from the UPC NPB suite. It computes the Fourier transform algorithm on a three-dimensional matrix using different domain decompositions, representing a widely extended code in scientific and engineering computing. This UPC kernel has been derived from the OpenMP FFT NPB implementation, and thus includes some significant changes on the variables with respect to the original Fortran code, in order to allow a better adaptation to the UPC syntax (e.g., user-defined data types are used to facilitate

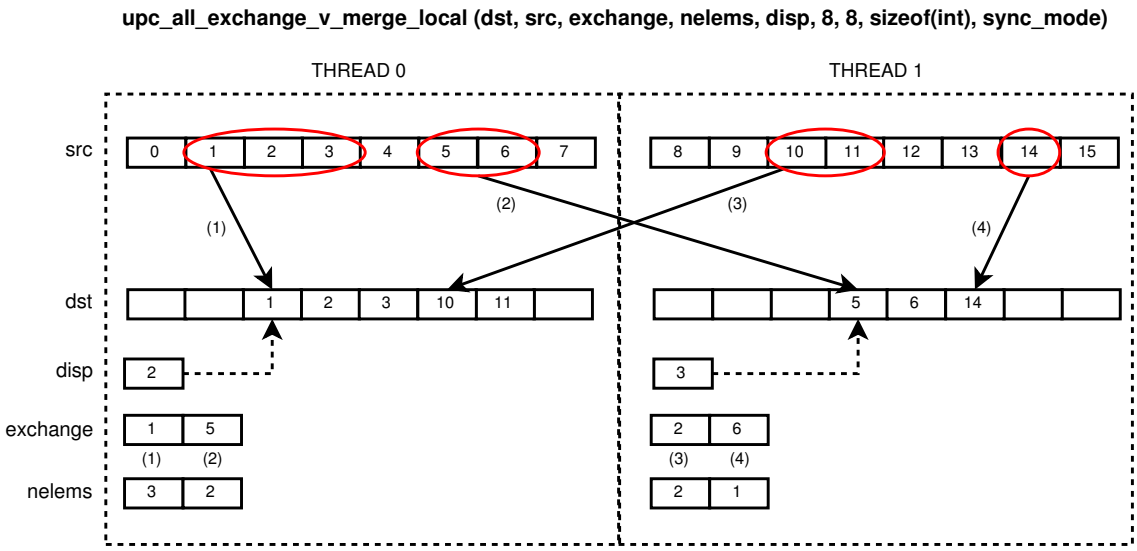


Fig. 3.11: Communications for `upc_all_exchange_v_merge_local` (2 threads)

the storage of complex values on each thread). The main computations of this kernel are performed in private memory using an array of structs (`u0`) that stores the initial conditions of the system in a linearized way analogously to the matrices of Section 3.4.1, and two working arrays (`u1` and `u2`) that assist the computation of the Fourier transform by storing intermediate calculations. The key part of this code is the computation of the transpose of the linearized matrix (stored in array `u1`) in `u2`, which is performed using a heavy all-to-all communication. Listing 3.19 shows the implementation of the remote communications used for the matrix transposition in the UPC FFT code, using `upc_memget` to obtain the corresponding array chunk from each thread: the data associated to a thread is stored in an array of complex values included as a member of a struct, which is defined for each thread in an array of shared structs with *THREADS* elements. This technique is used to avoid the definition of an array with a very large block size, which would affect performance.

```
for (i = 0; i < THREADS; i++) {
  upc_memget(
    (dcomplex *)&u2[MYTHREAD].cell[chunk*i],
    &u1[i].cell[chunk*MYTHREAD], sizeof(dcomplex) * chunk );
}
```

List. 3.19: Original UPC code in 3D FFT

Here the introduction of an extended exchange collective represents a better solution to implement all-to-all communications. However, the definition of the array of shared structs to store the data does not allow a direct application of this collective, as the source data is split. Therefore, some small changes are performed: the shared source array is referenced by a private pointer for each thread, and the *priv* variant of the exchange collective is applied here to obtain higher performance, as stated in Listing 3.20. Considering that the copy from `u1` to `u2` in this transposition is performed just to simplify the code, a second in-place solution is proposed, in which the all-to-all communication is performed on the same array `u1`. This approach only involves that the results of the communication are stored in `u1` instead of `u2`, which does not affect the final results of the FFT because the source data in `u1` is not reused after this communication. Both implementations with extended collectives, alongside with the standard approach, will be evaluated and tested in Section 3.6.

```
// First approach: u1 as src and u2 as dst
upc_all_exchange_priv(
    (dcomplex *)my_u2->cell, (dcomplex *)my_u1->cell,
    sizeof(dcomplex) * chunk, sync_mode);
// Second approach: in-place comms in u1
upc_all_exchange_in_place_priv(
    (dcomplex *)my_u1->cell, sizeof(dcomplex) * chunk, sync_mode);
```

List. 3.20: UPC code in 3D FFT using extended collectives

3.5. Microbenchmarking of UPC Collectives

The performance evaluation of the extended collectives library consisted of a microbenchmarking of representative collectives and an analysis of the impact of their performance on UPC kernels. Figures 3.12 to 3.14 show the microbenchmarking results for three extended collectives on two representative hybrid shared/distributed memory architectures. The selected collectives (or a variant of them) are used in the kernels presented in Section 3.4, therefore the analysis of their performance will help to illustrate the improvements that can be obtained in these codes by means of extended collectives.

The performance results have been obtained on two systems. The first one is the

JuRoPa supercomputer (from now on JRP) at Forschungszentrum Jülich (ranked 89th in the TOP500 List of November 2012), which consists of 2208 compute nodes, each of them with 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors at 2.93 GHz and 24 GB of DDR3 memory at 1066 MHz, and also an InfiniBand HCA (32 Gbps of theoretical effective bandwidth) with non-blocking Fat Tree topology. The second system is the Finis Terrae supercomputer (from now on FT) at the Galicia Supercomputing Center (CESGA), which consists of 142 HP Integrity RX 7640 nodes, each of them with 8 Montvale Itanium 2 (IA-64) dual-core processors at 1.6 GHz, 128 GB of memory and InfiniBand as interconnection network (4X DDR, 16 Gbps of theoretical effective bandwidth). On both systems, the UPC compiler is Berkeley UPC [3] v2.14.2 (with Intel icc v11.1 as backend C compiler) using its InfiniBand Verbs conduit for communications on InfiniBand. The Intel Math Kernel Library (MKL) v10.2 has also been used in the matrix multiplication kernels that will be evaluated in Section 3.6.

All the results shown here have been obtained using the UPC Operations Microbenchmarking Suite [45, 94]. Each figure presents data associated to a given extended collective, which has been executed using 32 threads on 8 nodes (4 threads per node). The message sizes considered for each collective range between 4 KB and 8 MB, showing aggregated bandwidth results for a clearer comparison against the bandwidth provided by the interconnection network (4 GB/s per link for JRP and 2 GB/s per link for FT). The identifiers for each entry in the legends have two parts: (1) the identifier for the implementation of the collective function, and (2) optional tags that identify a variant of the given implementation. The following list describes all the identifiers and tags that are used in the microbenchmarking figures:

- Identifiers:
  - `Original`: the corresponding standard implementation of the given extended collective (i.e., the one included in the standard UPC collectives library).
  - `Base UPC`: the implementation of the same functionality as the target extended collective using standard UPC facilities.
  - `Extended`: the actual function included in the extended collectives library, considering the optimizations presented in Section 3.3 in order to obtain

the best performance.

- Tags:
  - `priv`: the *priv* variant of the given extended collective (see Section 3.2.4).
  - `pvpr`: an implementation of a standard collective functionality that uses privatizations of shared variables whenever possible (see Section 2.1.1).

Figure 3.12 shows the performance results of the `upc_all_broadcast_in_place` collective, which has been used to implement the dense matrix multiplication presented in Section 3.4.1. In both test environments the best results correspond to the implementation of the extended collectives library. The *priv* version presents lower bandwidth because of the additional copy of the data from the private space to shared memory for each thread, which reduces around 12% the performance on JRP and 7% the performance on FT for 8 MB. In this latter scenario, the network communication is the main bottleneck, so this copy does not impact performance significantly. In both test environments the tree-based communications using a binomial-flat-tree algorithm (see Figure 3.10 for a reference) are playing the most important role in the optimization, as well as the efficient memory handling of the buffers for the *priv* case. As a result, the implementation with standard UPC facilities (`Base UPC`) generally presents the worst performance, except for message sizes larger than 256 KB in the FT testbed, where it outperforms the *priv* version because of the commented additional data copy. Regarding both testbed systems, the results of JRP clearly outperform those of FT, mainly because of the higher network bandwidth, thus in the following graphs only the results of JRP are shown for clarity purposes.

Figure 3.13 presents the bandwidth results of the `upc_all_scatter_v` collective, which is included in the implementation of the sparse matrix multiplication (see Section 3.4.2). The implementation of a vector-variant collective notably increases the complexity of the base collectives, therefore the bandwidth results are lower when compared to those in Figure 3.12. Regarding the comparison between algorithms, the extended collective outperforms the base UPC implementation in all cases. It is also important to remark the benefits of the privatization of array parameters when using the base UPC collective: using this feature, its implementation can obtain very important benefits, especially for large message sizes. However, the

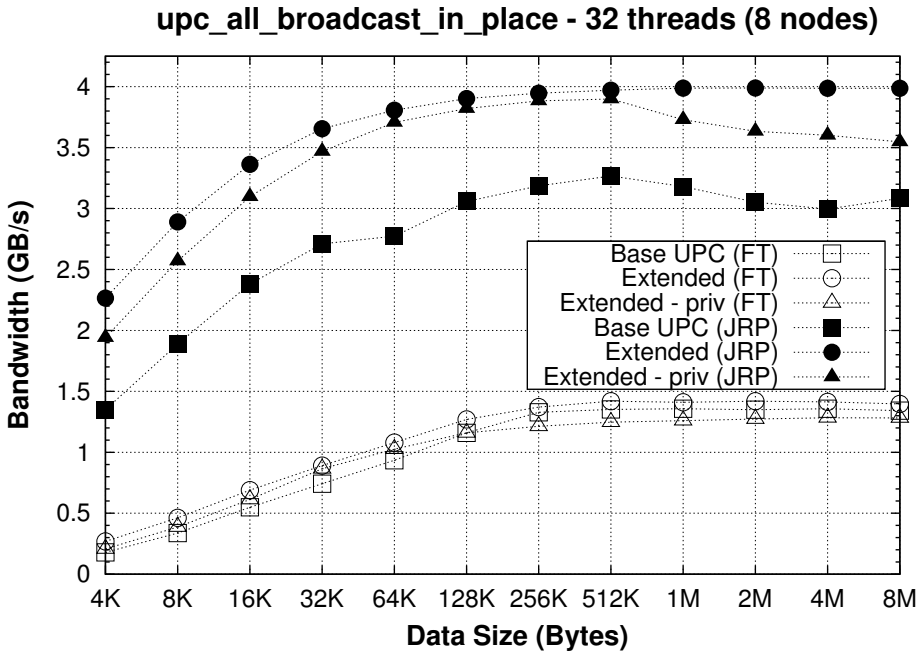


Fig. 3.12: Performance of `upc_all_broadcast_in_place`

tree-based algorithm is once again making the difference between the standard and the extended collective implementation.

Figure 3.14 shows the results of the `upc_all_exchange_in_place` collective, whose variants are used in the implementation of the Integer Sort and 3D FFT kernels (in Sections 3.4.3 and 3.4.4, respectively). Unlike the previous cases, the results of the original UPC exchange collective from the standard library is shown here, because its relatively low performance helps to illustrate how the extended in-place version is able to obtain higher bandwidth. Even the results of its *priv* version are significantly better than those of the original collective for message sizes between 128 KB and 1 MB, which indicates that not only the optimizations play an important role, but also a good algorithm is a key factor to obtain high performance (see Figure 3.3 for a graphical description). However, it is also significant that the `Base UPC` implementation (which uses the `Original` UPC exchange collective as a basis) obtains higher performance for message sizes smaller than 64 KB; therefore, the benefits of the new algorithm are effective when large messages are used.

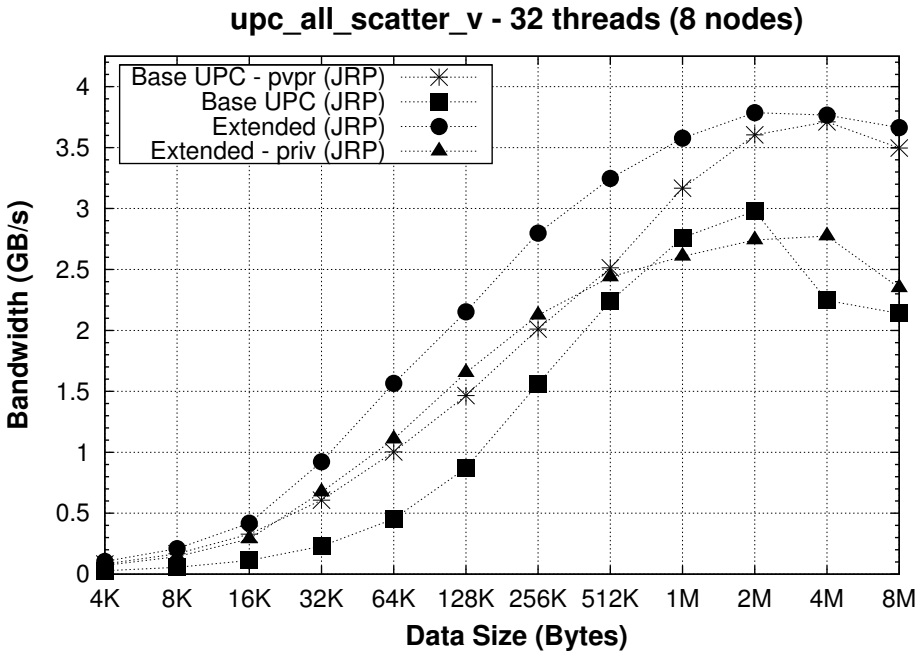


Fig. 3.13: Performance of `upc_all_scatter_v`

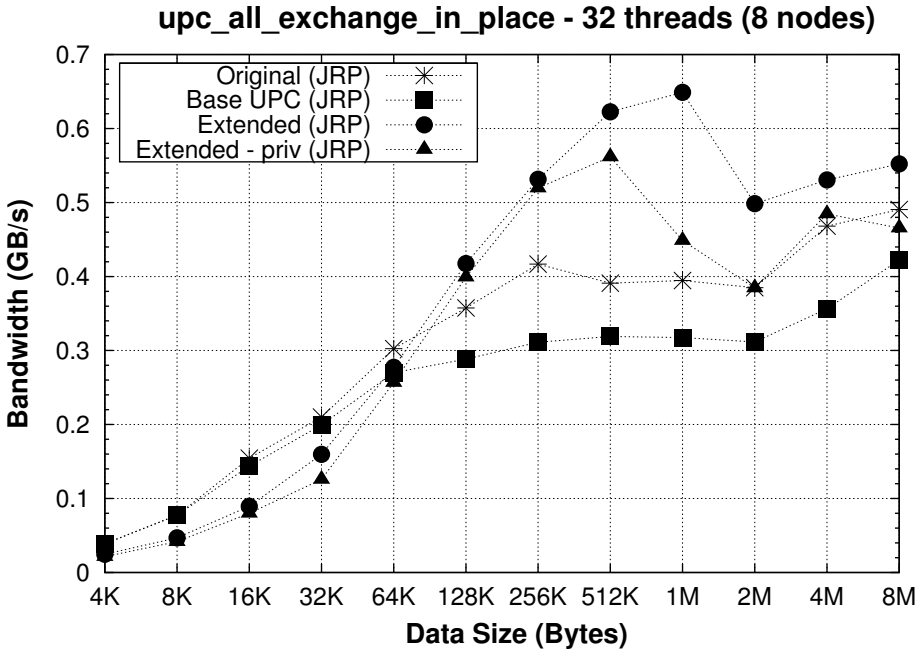


Fig. 3.14: Performance of `upc_all_exchange_in_place`

3.6. Performance Evaluation of UPC Kernels

This section presents a performance analysis of the representative collective-based UPC kernels discussed in Section 3.4 on the two testbed systems (FT and JRP) described in Section 3.5. The experimental results of the evaluated codes have been obtained using two different configurations of number of threads per node: (1) one thread per node, and (2) the maximum number of cores per node in each testbed system (fill-up policy), i.e. 16 threads for FT and 8 for JRP. In both configurations, the optimizations presented in Section 3.3 have been applied conveniently on both systems according to their requirements. Each of the analyzed codes has one implementation using standard UPC operations and an alternative version using the extended collectives library (see Section 3.4 for further details about its application to these codes). The standard UPC versions of NPB IS and FFT are available at [32], whereas the other codes that use standard UPC operations have been implemented following general guidelines for UPC hand-optimized codes [22].

Figure 3.15 shows the performance in terms of execution times and GFLOPS for the dense matrix multiplication of two 4480×4480 matrices of double precision floating-point elements, using a standard UPC code (labeled as "Standard UPC") and the extended collectives ("Extended Colls"). Here each UPC thread calls a sequential MKL matrix multiplication function, and all data movements associated to the workload distribution are performed by the collective functions. The results indicate that the use of extended collectives helps achieving the best results of this code in nearly all test cases, being specially advantageous for the worst scenario in the standard UPC code: when a single thread per node is used and thus the network overhead represents the main performance penalty. More precisely, the use of binomial-tree algorithms in executions with one thread per node favors a suitable exploitation of the internode bandwidth, whereas the use of intranode communications on shared memory with 8 (JRP) or 16 threads (FT) per node requires the binomial-flat-tree approach. In this latter case, the difference between the standard code and the extended collectives is smaller because of the maximization of flat-tree intranode communications. In general, the benefits of the use of extended collectives are noticeable especially when using 32 threads or more in both testbeds, where also the message segmentation improves cache data locality.

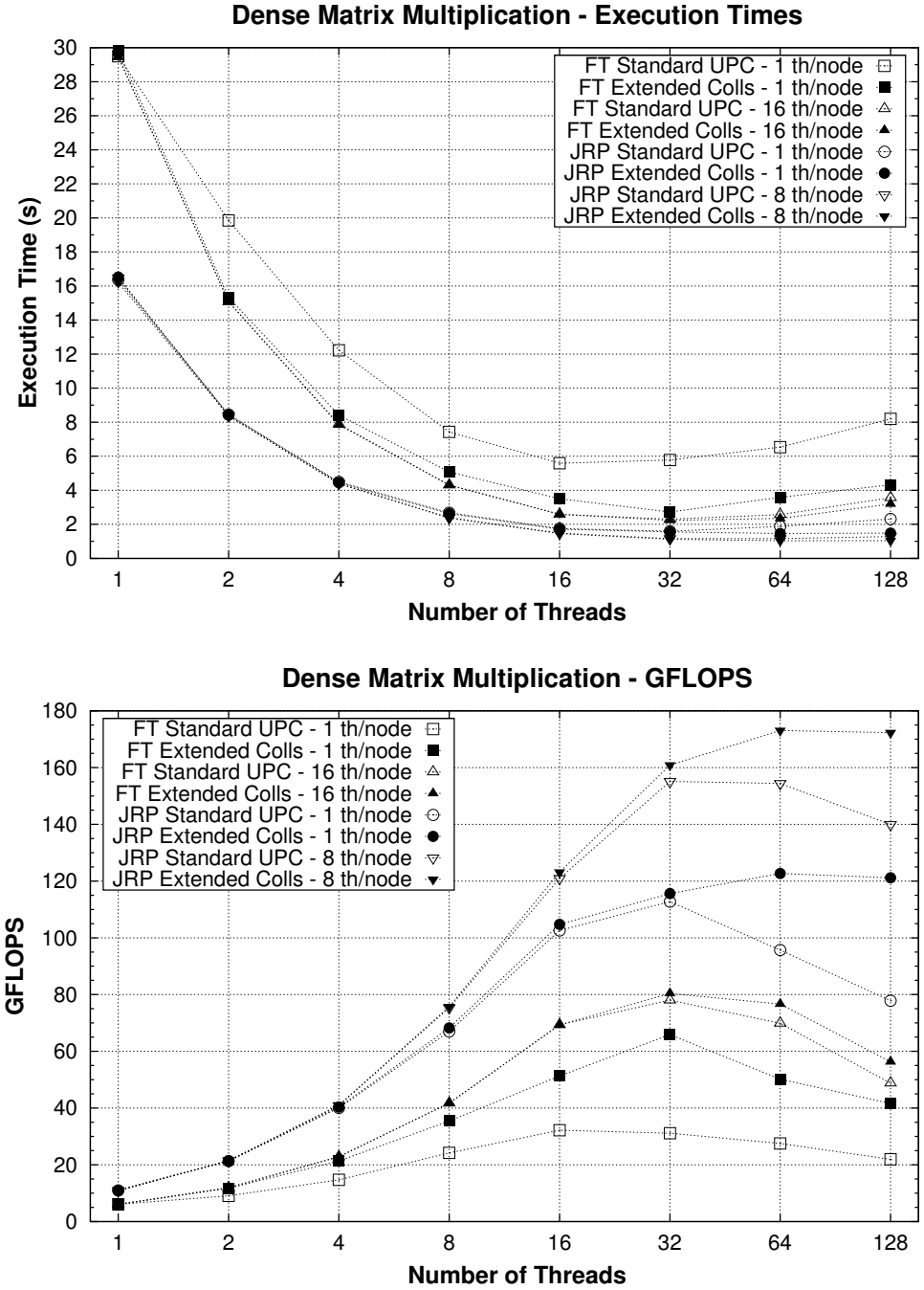


Fig. 3.15: Performance of dense matrix multiplication (4480×4480)

Figure 3.16 displays the performance of the sparse matrix multiplication code. The sparse matrix is a symmetric 16614×16614 matrix with 1,091,362 non-zero entries (0.4% of non-zero elements) generated by the FIDAP package in the SPARSKIT Collection [59], and the dense matrix has 16614×4480 double precision elements. Once again, the extended collectives provide better performance than the standard UPC code, especially as the number of threads increases. This improvement is a bit higher than in the dense case because of the larger amount of communications involved. As a result, the use of more efficient communication algorithms, which take advantage of hybrid shared/distributed memory architectures favoring data locality, is also a key factor in this scenario. Here the execution times are higher (the GFLOPS smaller) than for the dense case, which is due to the overhead of accessing the sparse matrix, in CSR format, as well as the communication penalty introduced by the data transfers required for this kernel (based on the vector-variant collectives), which makes the optimizations depend on a balanced workload distribution.

Figure 3.17 presents the performance of the standard version of NPB UPC IS in terms of execution times and millions of operations per second (Mop/s), compared to a version using the extended exchange collective (merge-local-get vector-variant, see Figure 3.11 for a similar example). For a small number of threads, the standard NPB code obtains similar results to the version using the extended collectives, although the difference grows as the number of threads increases, because of the positive effect of using binomial-tree algorithms for the extended collectives. It is also important to note that this kernel exploits intranode communications when using a single node on FT (up to 16 threads for a 16-thread configuration), whereas when using 32 or more threads the best results are obtained using a single thread per node, which is also the most efficient configuration for all tests in JRP. The IS kernel performs several all-to-all operations that exchange a significant number of messages between threads, and FT can take advantage of the use of intranode communications, but the contention in the access to the InfiniBand adapter on internode communications with 16 threads per node (in executions with multiple nodes) causes a significant performance drop. Regarding JRP, the use of a maximum of 8 threads per node and a better access to the network adapters allow better communication scalability.

Figures 3.18 and 3.19 show the performance results of different implementations of the 3D FFT kernel in terms of execution times and billions (10^9) of operations

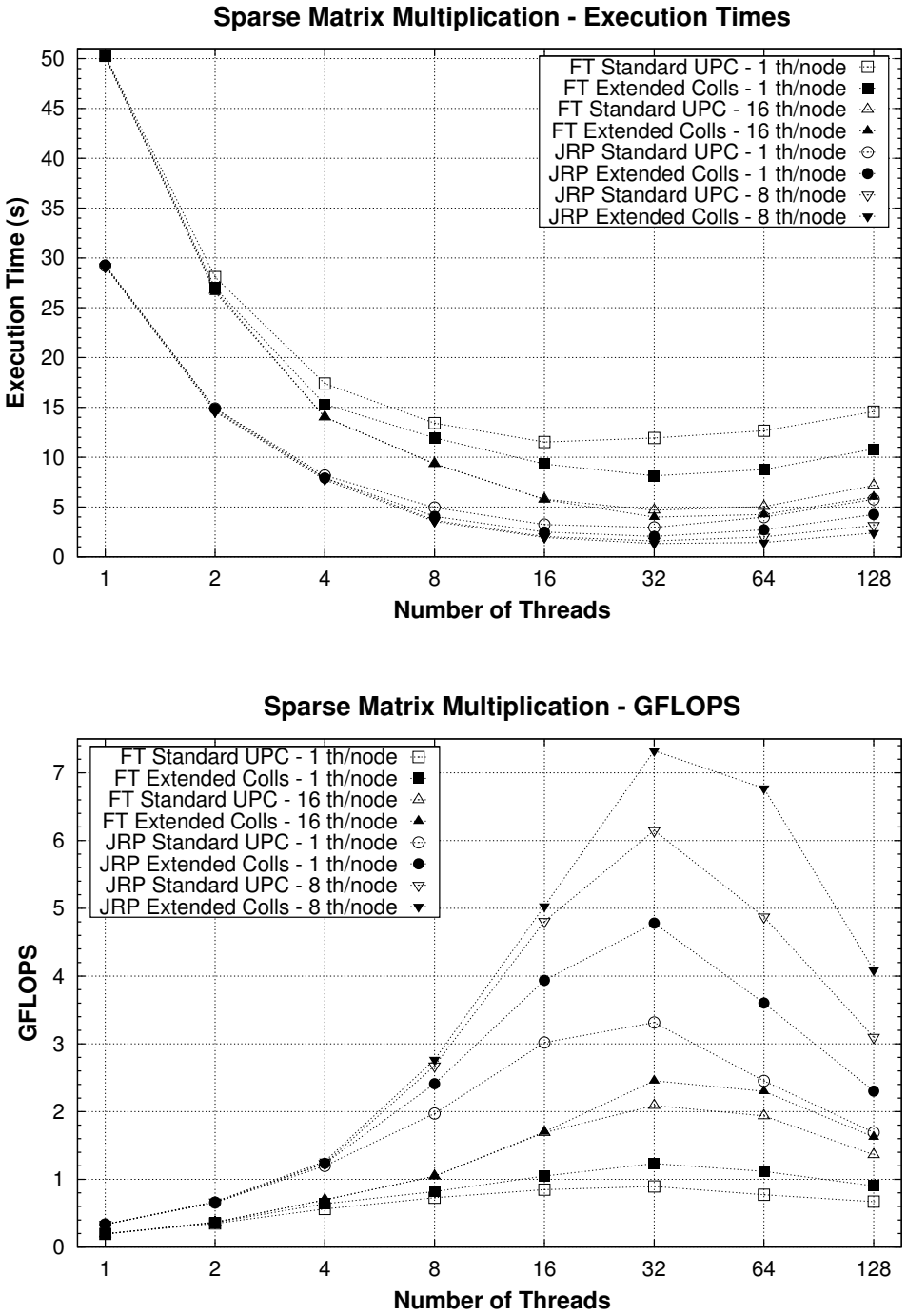


Fig. 3.16: Performance of sparse matrix multiplication (16614×16614 sparse matrix and 16614×4480 dense matrix)

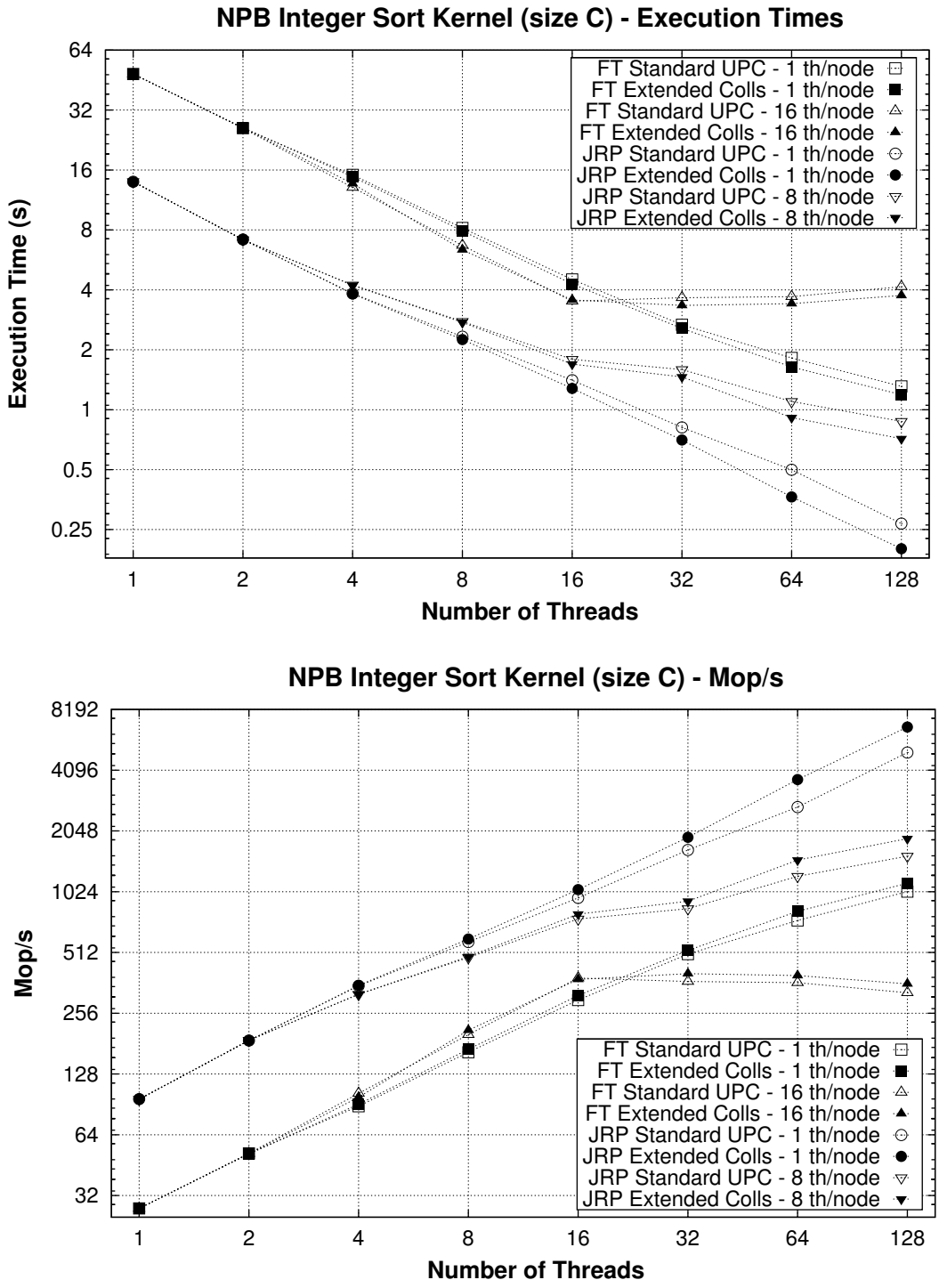


Fig. 3.17: Performance of NPB Integer Sort

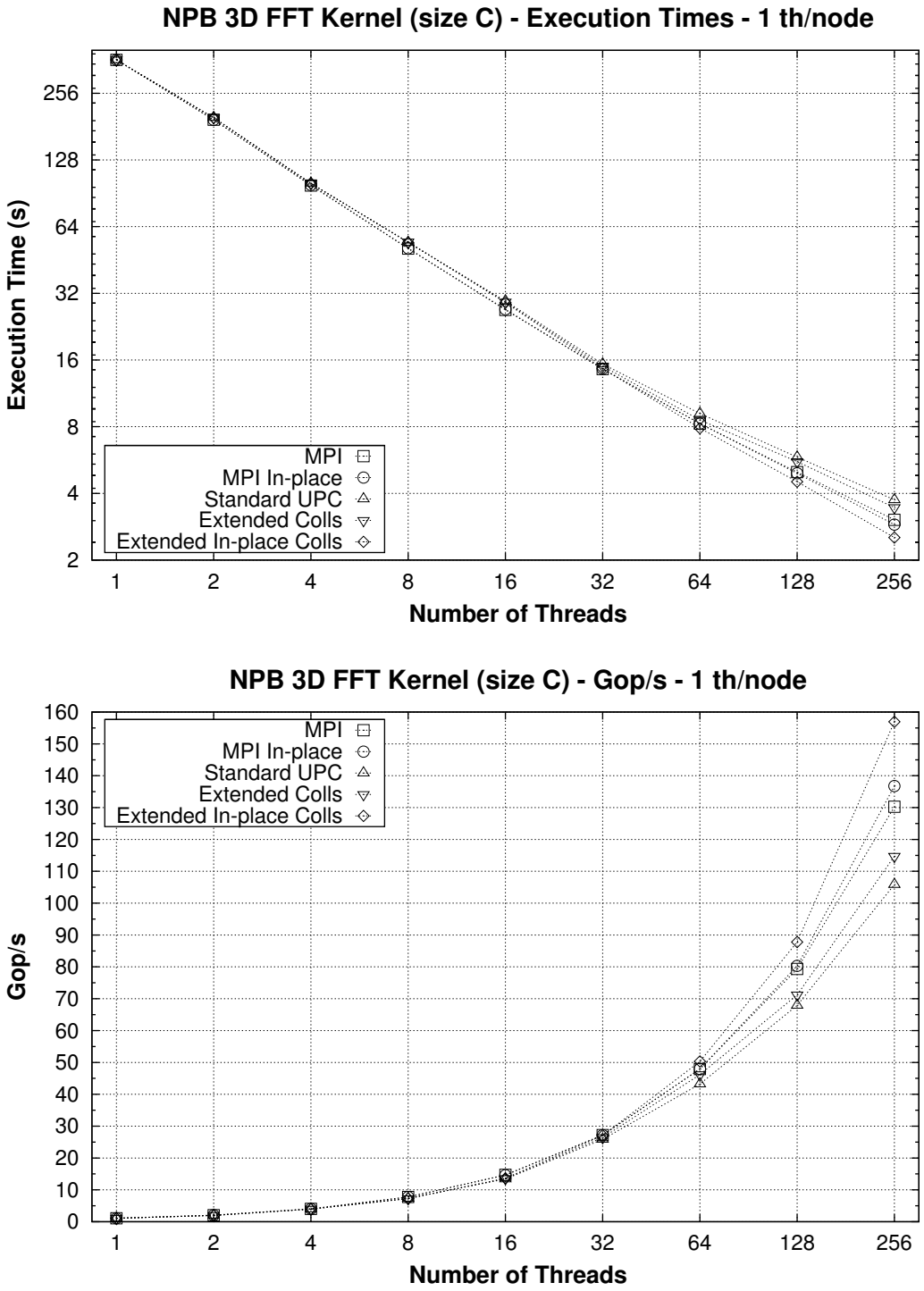


Fig. 3.18: Performance of NPB 3D FFT on JRP with 1 thread/node

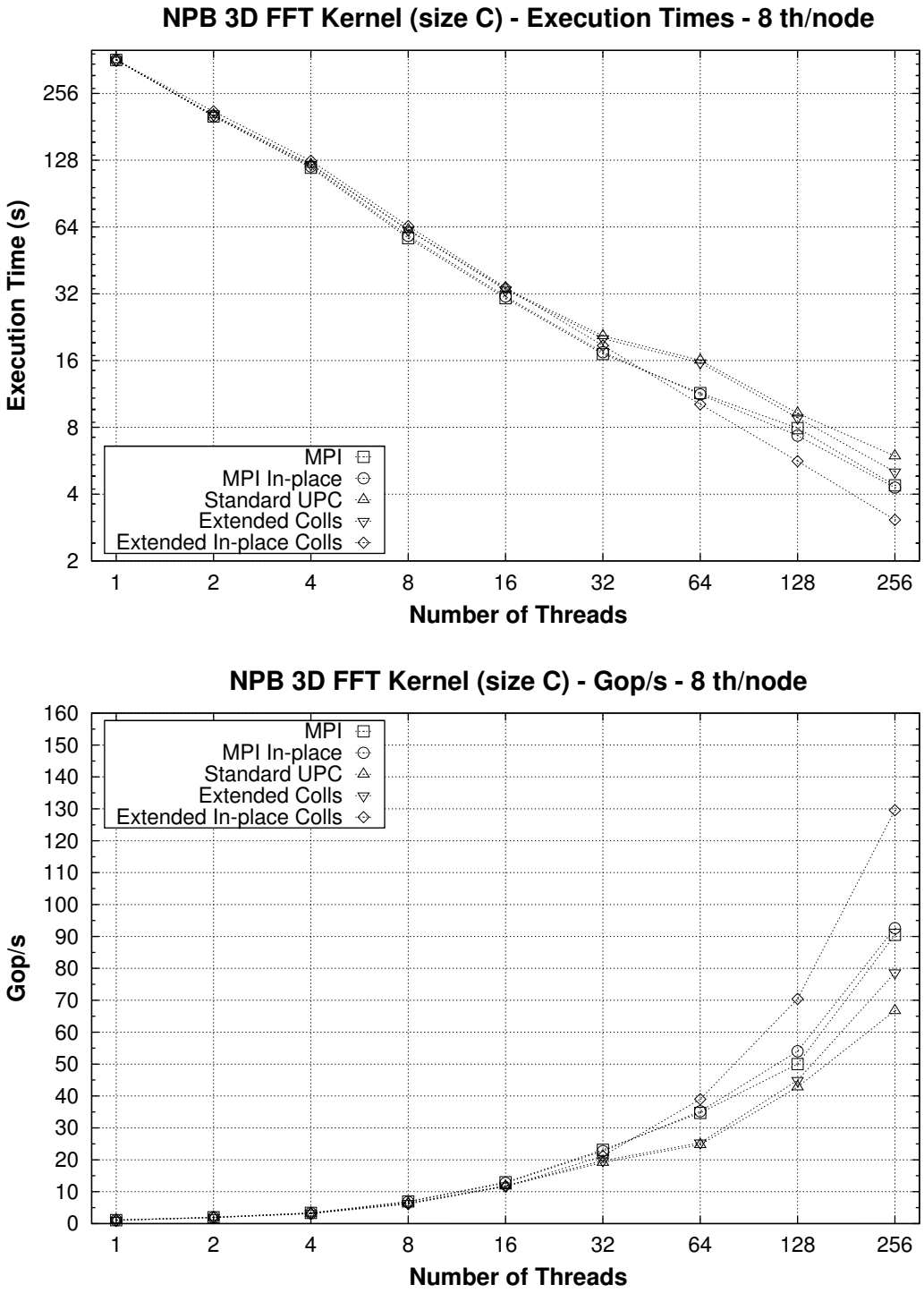


Fig. 3.19: Performance of NPB 3D FFT on JRP with 8 threads/node

per second (Gop/s) for 1 and 8 threads per node, respectively. In addition to the UPC codes described in Section 3.4.4 (the original standard UPC code, the one using a *priv* extended collective and the one using the *priv* in-place function), two more codes that use the C MPI library have been implemented in order to have a traditional parallel programming approach as a reference implementation for the UPC results: one code uses the standard `MPI_Alltoall` collective and the other uses the same collective with the `MPI_IN_PLACE` option, thus the array `u1` (see Section 3.4.4) is used both as source and destination. These MPI codes follow an analogous approach to the original UPC kernel to show a fair comparison with the UPC codes. As a result of the inclusion of the MPI code, and in order to favor the simplicity and readability of the graphs, Figures 3.18 and 3.19 only include the results obtained in the JRP system (where ParaStation MPI 5.0.27-1 [61] has been used as MPI compiler in the corresponding tests).

The comparison of these five codes gives out that MPI obtains slightly better performance up to 32 threads, but from then on the UPC in-place collective clearly presents the best results. The main reason is the use of the efficient algorithm that provides the best performance for large messages, as commented in Section 3.5 (see Figure 3.14), which is able to maximize the parallelism at a lower computational cost. Moreover, the higher the communication cost of the all-to-all, the better the *priv* in-place collective is able to perform, thus an execution with a high number of threads highlights the benefits of this implementation. Regarding the number of threads per node, the use of all cores in a node is worse than using only one for all UPC and MPI codes, which is due to the contention of InfiniBand communications similarly to the IS case.

3.7. Conclusions to Chapter 3

This chapter has presented the design, implementation, optimization and evaluation of a library of extended UPC collectives focused on providing higher programmability and overcoming the limitations of the standard UPC collectives library by enabling: (1) communications from/to private memory, (2) customized message sizes on each thread, and (3) the use of teams, among other features. The library consists of about 50 in-place, vector-variant and team-based collective func-

tions (not including the type variations of reduce, prefix reduce and allreduce), alongside versions of many of them (e.g., rooted) and *get-put-priv* functions, adding a total of more than 15,000 SLOCs (see Appendix A for the complete API). The algorithms implemented for these collectives are intended to maximize parallelism and efficiency by exploiting one-sided communications with standard memory copy functions. Moreover, an implementation at library level of teams has been developed to support team-based collectives, providing a general functionality that can be applied to any UPC code.

Four representative codes have been used for a comparative evaluation of the implemented library, and the results have shown that the use of the extended collectives has provided good solutions for all tested cases in terms of both performance and programmability. The extended collectives have been able to provide a more compact implementation of different communication patterns for the selected kernels. Moreover, the use of efficient collective algorithms enhanced performance for all the tests, especially for the 3D FFT code, in which the results have outperformed even the MPI counterpart (the UPC in-place collective obtained up to 28% of performance improvement for 256 threads). As a general outcome of the evaluation, these functions are able to obtain a better exploitation of computational resources as the number of threads and the amount of data to be transferred increases. In summary, these collectives provide a powerful and productive way for inexperienced parallel programmers to implement custom data transfers and parallelize sequential codes, as well as a wide variety of resources for expert UPC programmers, that can transparently take advantage of the optimizations implemented in this library.

After presenting some examples of computational kernels that use collective functions, the next chapters of the Thesis will focus on large-scale applications: (1) a UPC MapReduce framework and (2) a Brownian dynamics simulation.

Chapter 4

MapReduce for UPC: UPC-MR

This chapter presents the analysis, design, implementation and evaluation of a UPC MapReduce framework: UPC-MR [81]. The implementation of coarse-grain parallelism to process large data sets in a distributed environment has become a very relevant issue nowadays, and here the use of the MapReduce paradigm has proved to be a suitable solution in a wide variety of environments and applications. For example, MapReduce processing approaches are being applied to information retrieval [51], pattern recognition [42] or processing of medical information [50].

The motivation of this implementation is to put together the processing power required by MapReduce workloads and the programmability and efficiency of PGAS languages in order to manage current and upcoming hardware architectures. This framework will present the applicability of the PGAS UPC syntax in the development of a high-level data structure that facilitates abstract processing and a specific syntax for code parallelization with UPC, which will give an answer to the requirements detected in the programmability sessions of Section 2.2. Additionally, this code will show the application of some collective functions from the extended library described in Chapter 3, presenting performance results up to 2048 cores.

The following sections begin with a brief description of the state of the art in MapReduce (Section 4.1). The design and implementation of the UPC-MR framework is presented in Section 4.2, and the performance results using four representative applications are analyzed in Section 4.3, studying the impact of different processing approaches and the use of extended collectives.

4.1. Introduction to MapReduce

The MapReduce processing consists of two stages: the "Map" stage, in which a function is applied in parallel to a set of input elements to generate another set of intermediate key/value pairs, and the "Reduce" stage, in which all the intermediate pairs with the same key are combined to obtain the final output. These ideas were originally implemented in functional programming languages, but currently, following the guidelines of the MapReduce framework published by Google [12, 13], many other implementations of this framework using different languages have been developed. The use of MapReduce has been generally applied to large data-parallel problems that deal with the processing of large sets of files, and this kind of problems have also been used as testbed for performance evaluations [66]. As these tasks are usually integrated in applications written in object-oriented languages, most of the existing MapReduce implementations have been written using those languages, such as Google MapReduce (C++) or Apache Hadoop (Java) [33]. Additionally, there is also a recent work on the implementation of MapReduce for the X10 programming language [18], which uses the PGAS model as basis for parallelism, focusing specifically on exploiting programmability.

Nevertheless, there is still very little work on MapReduce applied to HPC. There are some interesting proposals of MapReduce for HPC, implemented in C on shared memory, such as the Phoenix project [102] and Metis [48], or using MPI C (linked to a C++ library) for distributed memory environments, like MapReduce-MPI [63]. Additionally, a study of the possibilities of implementing an optimized version of MapReduce with MPI [36] concluded that it is feasible, but additional features to favor productivity and performance in MPI, such as improved collective communications, are needed. At this point, UPC-MR has been developed aiming to provide a useful HPC implementation of this processing, favoring its integration on existing applications and offering good performance on multicore systems.

4.2. Design and Implementation of UPC-MR

The UPC-MR code is structured in two types of functions: the generic management functions and the user-defined functions. Management functions are used to

support the MapReduce functionality, that is, they initialize the framework and perform the necessary communications between threads for work distribution at "Map" and data collection at "Reduce". User-defined functions include the processing that should be performed on each element at the map stage and on the intermediate elements at the reduction stage. In order to maximize simplicity and usability, some basic ideas have guided the implementation of the UPC-MR framework:

- No parallel code development is needed: the management functions can perform all the parallel work, thus the user can simply write the sequential C code for the *map* and *reduce* functions that will be applied to the input and intermediate elements, respectively. However, if more control on the work distribution at the "Map" stage is desired, the framework also allows the user to disable all these mechanisms and define a custom parallelization routine within the *map* function.

- Simplicity in management: the generic management functions for MapReduce should also be written in an expressive and simple way. If the user needs to modify them, the use of a clear coding favors a better understanding of the processing of the management functions. Traditionally, UPC code has always tried to exploit performance by using privatizations and direct data movements using bulk memory copies (see Section 2.1.1 and reference [22]), but here the UPC-MR management codes tend to use higher-level constructs, such as collective functions, that encapsulate data movements between threads.

- Reusability and flexibility: the tasks implemented by the management functions are kept as generic as possible. Thus, the parameters to these functions are treated as a homogeneous set of values, in which each value has to be processed in the same way, regardless of its type. This generic typing is obtained by means of arrays of void type (void *), and the correct interpretation of input and output data to these functions is left to the user, because it can vary depending on the application in which MapReduce is used.

The piece of code included in Listing 4.1 presents the signatures of the two user-defined *map* and *reduce* functions in this framework (`mapfunc` and `reducefunc`, respectively), as well as the MapReduce management functions that perform the

mapping (`ApplyMap`) and the reduction (`ApplyReduce`). The next subsections give a more detailed description of the implementation process of the latter two, and also some general remarks on the UPC implementation of the framework.

```
void *mapfunc(void *input, void *key, void *value);
void *reducefunc(void *key, void *value, int nelems, void *result);
int ApplyMap(
        int (*mapfunc)(void *,void *,void *), void *inputElems,
        int nelems, int userDefDistrFlag, int algorithm, int *weights
);
void *ApplyReduce(
        int (*reducefunc)(void *,void *,int,void *), int nelems,
        int gathFlag, int collFlag, int sizeKey, int sizeValue
);
```

List. 4.1: Signatures of the basic functions in UPC-MR

4.2.1. Function `ApplyMap`

As can be seen in Listing 4.1, this function receives six input parameters: (1) the function that should be used for the "Map" stage, (2) the list of elements to which it has to be applied, (3) the number of elements in that list, (4) a flag (`userDefDistrFlag`) that indicates whether the work distribution is performed by `ApplyMap` or the user defines a custom distribution, (5) an integer that identifies the distribution algorithm, and (6) an array of weights that indicates the workload assigned to each input element in the list. According to this interface, first each thread must have the whole list of input elements stored in its private memory before calling `ApplyMap`, and then the list is split in subsets of elements that should be processed by each thread. The work distribution flag determines the type of processing used in `ApplyMap`. If this flag is not enabled, `ApplyMap` distributes the input elements between threads according to the algorithm selector passed as parameter, and the *map* function provided by the user is applied to each single element to generate an intermediate key/value pair (from now on, this will be referred to as "element-by-element map"). If the flag is enabled, the following two parameters in `ApplyMap` are ignored, and the work distribution routine is performed inside `mapfunc` for the whole set of input elements ("array-based map"), thus returning a set of intermediate elements. As a result of this, the implementation of `mapfunc` (defined by the user) must be consistent with the work distribution selected in `ApplyMap`. When an element-by-

element processing is chosen, `ApplyMap` implements four different algorithms, that are selected using the following values for the `algorithm` parameter:

- *UPC_MR_BLOCK*: this algorithm divides the number of input elements according to a block distribution.
- *UPC_MR_CYCLIC*: this option assigns each input element to a thread in a cyclic (round-robin) way, according to the element position in the input array and the thread identifier. The block and cyclic algorithms do not require the definition of the `weights` parameter.
- *UPC_MR_BALANCED*: this algorithm implements a heuristic to obtain a load balanced distribution of input elements between threads according to the values passed as parameter in the `weights` array (positive integer values). A high value for a weight indicates that the associated input element presents a high workload, thus each element in the input vector is assigned to the thread whose current associated workload (according to the value defined in a counter) presents the lowest value of all threads.
- *UPC_MR_BEST_FIT*: this option indicates the selection of the most efficient algorithm among the three previous candidates. To do this in a compact way, the first element in the array of weights represents here a threshold value, and the weights associated to the input elements are stored in the following array positions. The threshold value indicates the maximum difference between workloads for different threads that may be considered as acceptable in order to have a load balanced execution. The workloads for the block and cyclic algorithms are computed, and if the highest workload difference is below the threshold for one of them, that algorithm is selected for work distribution (if both pass the test, the block distribution is selected by default). If none of these is suitable, the balanced algorithm is selected.

Figure 4.1 presents an example of work distribution among three threads using a set of 9 input elements with different computational weights, with the maximum difference of workload between threads. Here the *BALANCED* algorithm obtains a more similar distribution of workload between threads than the *BLOCK* or *CYCLIC*

ones, because it takes advantage of the implemented heuristic. Consequently, for this input set the *BEST_FIT* algorithm would select the *BALANCED* distribution.

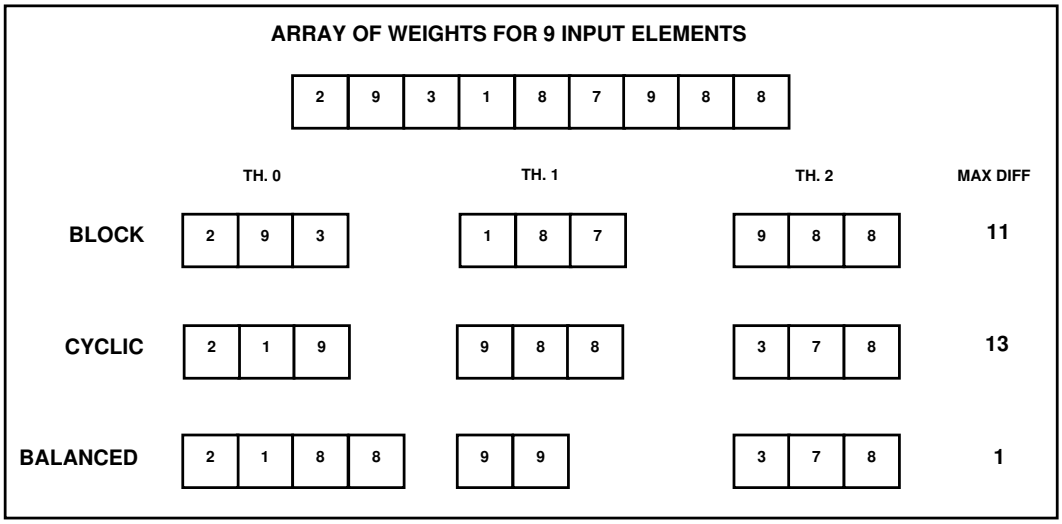


Fig. 4.1: Examples of *BLOCK*, *CYCLIC* and *BALANCED* work distributions

It is important to remark that after the execution of `ApplyMap` each thread keeps its own sublist of intermediate elements stored in the private memory space, which is completely independent from the rest of the threads. These intermediate elements are managed by the MapReduce framework transparently to the user, thus they do not need to be returned by `ApplyMap`.

4.2.2. Function `ApplyReduce`

This function takes six parameters (see Listing 4.1): (1) the function that will be used to combine the elements produced by each thread in the "Map" stage, (2) the number of intermediate key/value pairs per thread, (3) the gathering flag, (4) the collective communications flag (`collFlag`), (5) the size of each intermediate key, and (6) the size of each intermediate value. The gathering flag (`gathFlag`) plays an important role in deciding the necessary communications between threads at the beginning of `ApplyReduce`. If its value is *UPC_MR_NOCOMM*, it indicates that each thread only requires its intermediate values to return the final result, thus no communications are performed. When set to *UPC_MR_ALLCOMM*, all threads should gather the intermediate data from the rest of the threads in order to compute

the reduction. Otherwise, a positive integer value from 0 to *THREADS*-1 indicates the identifier of the only thread that gathers the intermediate data and computes the final result.

The communications required here by `ApplyReduce` are performed at the same time by all threads, so collective functions can be used. However, UPC-MR cannot apply here directly the standard UPC collective primitives, for two major reasons: (1) MapReduce operates on each thread's private memory space, but standard UPC collectives must use shared arrays as source and destination, and (2) some collective communications, such as the gathering of intermediate elements, generally have a different number of elements per thread, which is not supported by the standard UPC collectives. Thus, extended collectives are necessary at this point.

The use of extended collectives in `ApplyReduce` is controlled using the collective communications flag (`collFlag` parameter in Listing 4.1). If this flag is enabled, collective functions are used for communications, and the user should indicate the size of each key and value in the set of intermediate elements as parameters to `ApplyReduce`. The collective call used here is the *priv* version of the vector-variant allgather collective (see Section 3.2.2). However, if all intermediate elements are not equal in size, the `sizeKey` and `sizeValue` parameters are ignored and the *raw* version of the previous collective is used in order to indicate the amount of raw data that is transferred to each thread. In this case, the MapReduce framework detects this situation internally when generating the output of `ApplyMap`, and therefore the system builds indexes to keep track of the start position of every intermediate element to reconstruct them and perform the reduction. The memory consumption of this procedure can be high but, in general, this is not the most common situation, as all intermediate keys and values typically have the same size. When `collFlag` is not enabled, the intermediate elements are transferred one by one to their destination(s).

After the data communication step in `ApplyReduce` (if required), the *reduce* function defined by the user (`reducefunc`) is applied to the set of intermediate elements stored in the private memory space of each thread. This user-defined function receives as input parameters two arrays of keys and values (that represent the set of intermediate elements) and the number of elements. The combination of all these intermediate elements is considered as the final result of MapReduce, and it is returned by one or all threads. If no communications between threads took

place in the previous step, each thread returns a portion of the final result, and if communications were performed, all threads (or only the selected thread) return the complete final result.

The implemented approach establishes that `reducefunc` must always process all the intermediate elements associated to each thread in a single call, analogously to the definition of the array-based map in `ApplyMap` commented in Section 4.2.1. This design decision allows a more flexible definition of customized reduction functions for the user. An alternative design for `ApplyReduce` could have also allowed the use of an element-by-element reduction, but this type of processing would only be useful for a restricted set of reduction operations on basic datatypes (e.g., a sum of integers or a logical AND operation), and it would be difficult to perform many other operations. For example, the computation of the average value in a list of integers would imply either the definition of a variable number of parameters for the element-by-element reduction (which would complicate the design of the function and thus cause unnecessary trouble to the user), or the transfer of the essential part of the processing to the `ApplyReduce` function (which would not be acceptable for abstraction purposes). Therefore, the definition of the reduction on the complete set of intermediate elements has been considered here as the best choice.

4.3. Performance Evaluation

This section includes performance results for UPC-MR on shared memory (using the Phoenix system [86] as reference for comparison) and distributed memory (compared to the MapReduce-MPI framework [49]). Both frameworks rely on C++ libraries, but the MapReduce codes developed with them can be written in C (using the MPI library for MapReduce-MPI), therefore they can offer better performance than other approaches that use Java codes, as stated in previous evaluations [66].

4.3.1. Experimental Conditions

In order to evaluate the MapReduce framework, four applications have been selected: (1) Histogram (*Hist*), (2) Linear Regression (*LinReg*), (3) Word Count

(*WordC*) and (4) Spam Count (*SpamC*). *Hist* obtains the values of the pixels in a bitmap image (values for RGB colors, ranging from 0 to 255) and computes the histogram of the image, that is, it counts the number of pixels that have the same value for a color. *LinReg* takes two lists of double precision paired values that represent coordinates, and computes the line that fits best for them. *WordC* processes a set of input text files in order to count the number of occurrences of distinct words. *SpamC* is similar to *WordC*, but it processes a set of email spam in plain text and counts the occurrences of 256 words related to malware.

The UPC codes for *Hist* and *LinReg* are based on the ones included in the Phoenix distribution, and although some changes have been made in order to perform representative tests (e.g., the input values to *LinReg* used for this performance evaluation are double precision numbers), the codes are designed to provide a fair comparison for UPC, C and MPI. It is important to note that, according to the approaches at the "Map" stage described in Section 4.2.1, Phoenix always performs an array-based map, whereas MapReduce-MPI always uses an element-by-element map; thus, the UPC code has been adapted to have a fair comparison with both approaches. Additionally, the work distribution for all tests has been implemented using a block algorithm, and collectives are used for all codes. The input images for *Hist* were obtained from the Phoenix web, whereas the *LinReg* coordinates were randomly generated. Regarding the files for *WordC*, they were taken from the SPAM Corpus in the TREC Conference collection [75], and the set of email spam for *SpamC* was obtained from the Webb Spam Corpus [98]. Both corpora are widely used in tests with similar codes, especially in the area of information retrieval.

The UPC-MR implementation has been evaluated using three different testbeds. The first one (named SMP) is a multicore system with 2 Intel Xeon E5520 processors with Simultaneous Multithreading (SMT) enabled (8 cores per node which are able to run 16 threads simultaneously) and 8 GB of memory. The second one is FT, the Finis Terrae supercomputer installed at the Galicia Supercomputing Center (CESGA), and the third system is JRP, the JuRoPa supercomputer at Jülich Supercomputing Centre, which has been used in order to obtain results up to 2048 cores. See Section 3.5 for more details on the latter two systems. The UPC compiler used was BUPC (v2.12.1 of December 2010 was used in SMP and FT, whereas v2.14.0 of November 2011 was used in JRP), with Intel icc v11.1 as C background

compiler for all environments. BUPC uses threads for intranode communications and InfiniBand Verbs transfers for internode communications. The HP MPI 3.3-005 library was used in FT by MapReduce-MPI, and ParaStation MPI 5.0.27-1 [61] was used in JRP. All executions on distributed memory maximize the number of UPC threads (or MPI processes) per node, i.e. using up to 16 and 8 cores per node in FT and JRP, respectively.

4.3.2. Performance Results

Figure 4.2 presents the execution times on distributed memory (for InfiniBand communications) using UPC and MPI for *Hist* and *WordC*. The top graph shows that the execution times of UPC are clearly lower than those of MPI for *Hist* using 104.53 millions of input values, but in the bottom graph the *WordC* execution times with 75,419 input files are very similar for UPC and MPI, achieving both implementations a speedup of about 75 for 128 threads. The reasons for this behavior are the amount of input elements used in each code and the implementation of MapReduce support on MPI using C++ libraries. A profile of MapReduce-MPI indicates that 90% of the sequential execution time of *Hist* is spent in handling the C++ library structures associated to every input element, whereas the percentage falls to an 8% for the presented *WordC* problem size (values obtained from a performance profiling). Therefore, when there is a very large number of input elements with little computation (value testing and clustering in *Hist*), the element-by-element map processing performed by MapReduce-MPI involves much more overhead (and hence a significant impact on the overall performance) than processing a reduced number of input elements with more intensive computation for each one (as for *WordC*). Nevertheless, the element-by-element map in UPC-MR is not affected by this circumstance because the commented C++ library calls are not used. As a consequence of this, MPI results are not shown in the following figures for *Hist* and *LinReg*, because they process a large number of input elements and MapReduce-MPI performs poorly in these scenarios.

Additionally, the top graph of Figure 4.2 shows that the use of an array-based map with UPC obtains better results for *Hist* than the default element-by-element map in `ApplyMap`, because the array-based approach allows the user to define an

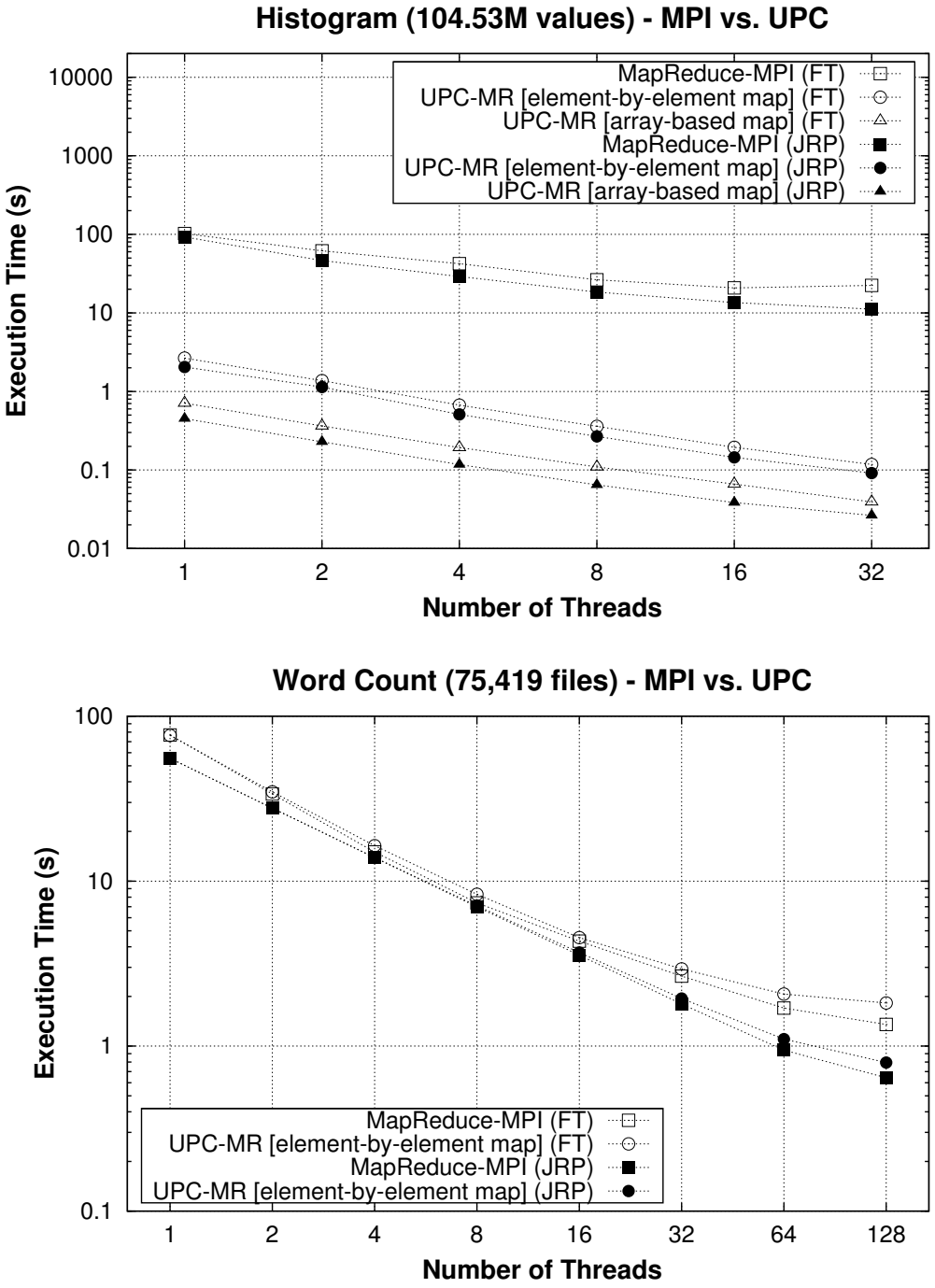


Fig. 4.2: Performance comparison of MapReduce-MPI and UPC-MR on distributed memory

optimized *map* implementation for the whole set of input data, and not just for every single element separately. Regarding the *WordC* code, these differences are negligible, therefore the results obtained using an array-based map are not shown in the bottom graph. The possibility of implementing an optimized array-based map for a given code greatly depends on the nature of the input data. For instance, the optimization performed here for the *Hist* code consists in reading all input data as a single array of characters, distributing it in *THREADS* balanced chunks (one chunk per thread) and classifying the elements using bytewise comparison. This procedure can be implemented because all input elements in *Hist* have the same size (`sizeof(int)`), but it is impossible to do the same optimization for *WordC* because the words have different length. As a result of this, when the use of an optimized array-based map is possible, it is the best option to obtain the best performance. However, for a low or medium number of elements to be processed, such as in *WordC*, the programmers can safely rely on the default element-by-element map, which does not require the implementation of UPC routines for data distribution.

Figure 4.3 shows performance results for *Hist* using 468.75 millions of input values. In the shared memory graph, both testbeds are using SMT for executions up to 16 threads (which limits the speedup of the codes), and in general the UPC implementation of *Hist* has better performance on shared memory than Phoenix. This is related to the amount of input elements processed and the use of shared C++ objects in Phoenix to process the intermediate data, which involves a higher overhead than the UPC processing. Unlike MapReduce-MPI, Phoenix has similar execution times than UPC-MR, mainly because of the use of an array-based approach at the "Map" stage. Regarding distributed memory (InfiniBand communications), the bottom graph presents the execution times for the UPC implementation in order to show that the array-based approach is very relevant to obtain lower execution times, because of the use of the optimizations commented for the top graph in Figure 4.2.

Figure 4.4 presents the results of the *LinReg* code using 60 millions of input values. The top graph confirms that UPC presents slightly better results than Phoenix for *LinReg* on shared memory. Here the differences between implementations are smaller than in the *Hist* graphs because the amount of input elements is also smaller and thus the Phoenix C++ library overhead is reduced. The execution times for *WordC* on shared memory (not shown for simplicity) follow the same trend: they

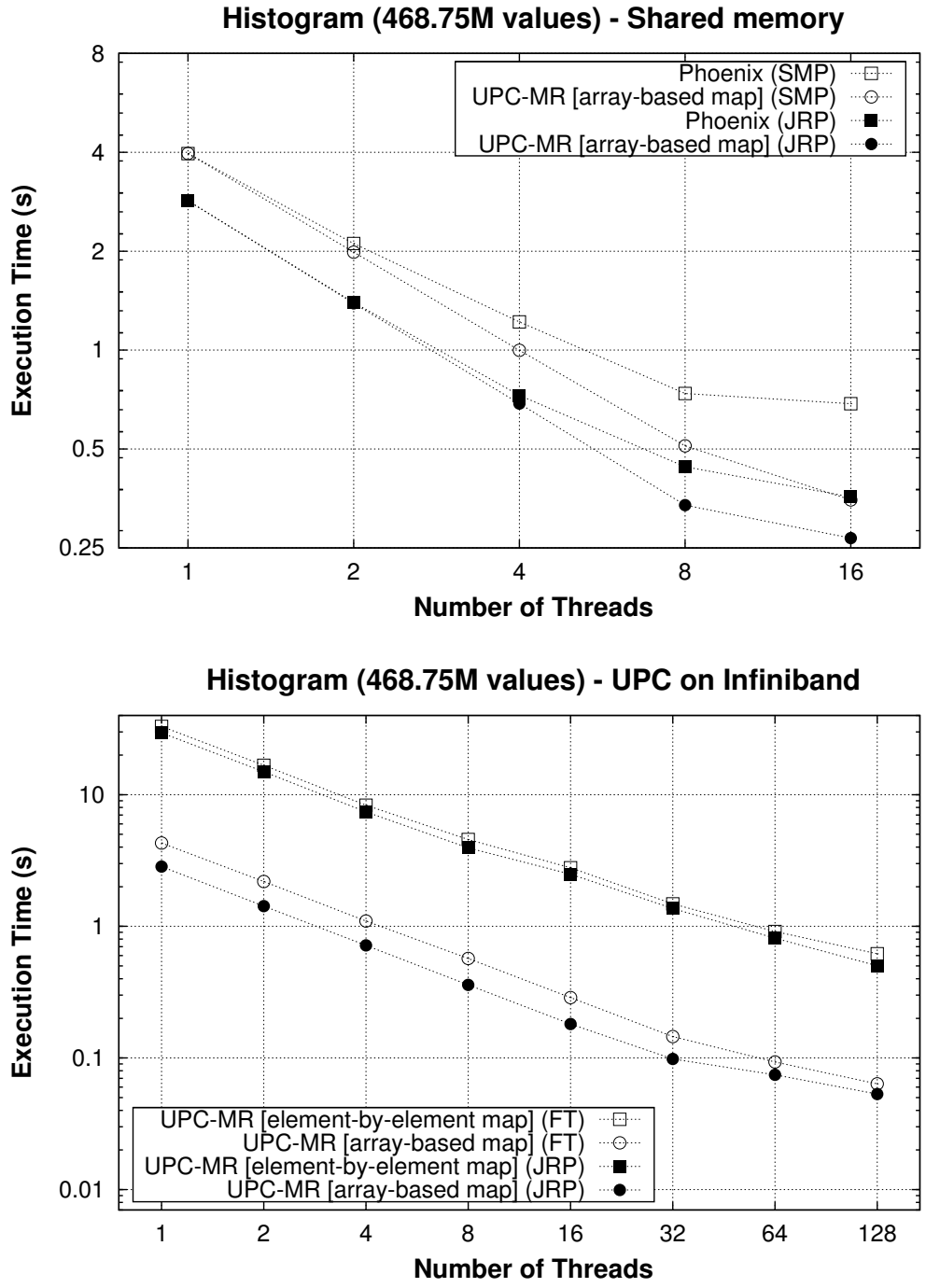


Fig. 4.3: Performance of *Hist* on shared and distributed memory

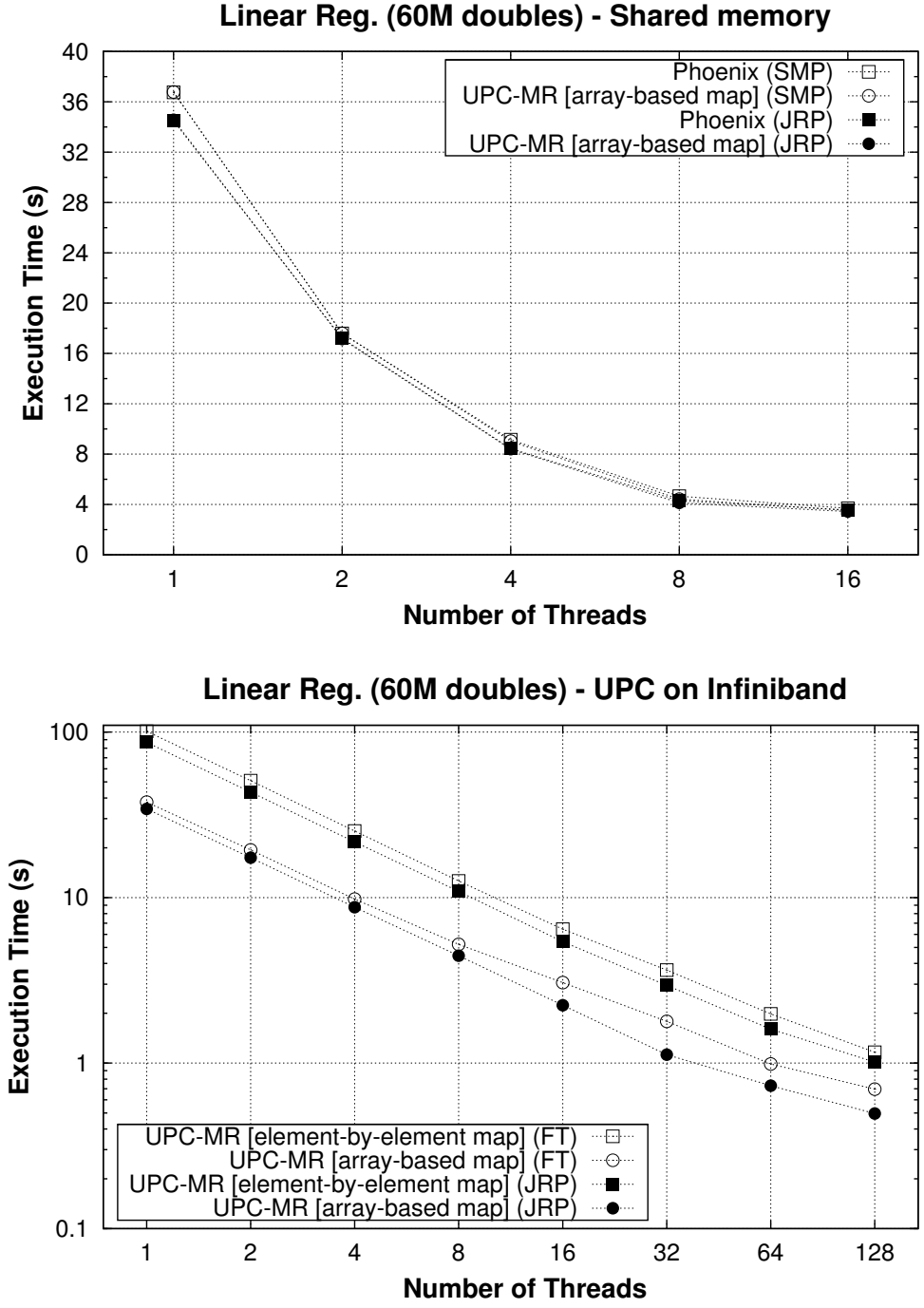


Fig. 4.4: Performance of *LinReg* on shared and distributed memory

are almost similar for Phoenix and UPC (tested with the whole TREC corpus, 75,419 files). Once again, the bottom graph of Figure 4.4 shows that the use of an array-based map is a key factor to obtain better performance.

Figure 4.5 presents the performance results of UPC-MR (execution times and speedups) with a large number of threads for the *SpamC* application using 100,000 input files from the Webb Spam Corpus, and for *LinReg* using 1 billion (10^9) of paired values. These results are intended to show that both implementations are able to scale up to a large number of threads: the large computational power and high scalability provided by the JRP system, as well as the small weight of `ApplyReduce` in the total execution time of both applications, provide a favorable scenario. However, similarly to the previous codes, the size of communications in `ApplyReduce` for both applications increases proportionally with the number of threads. Therefore, the speedup tends to decrease for a large number of threads when the execution time of the reduction part becomes relevant when compared to `ApplyMap` (i.e., generally when dealing with reduced workloads).

Additionally, these codes have been used to study the impact of the use of extended collectives in the UPC-MR framework: an analogous standard code to implement these collectives is compared to the corresponding extended collective call, both when using 1 and 8 threads per node. As a result of the low time consumption of `ApplyReduce`, the differences between the standard communications and the extended collectives are only noticeable for more than 64 threads in *SpamC*, and even for more threads in *LinReg*. In both cases, the extended collectives have been able to exploit the use of hybrid shared/distributed memory communications, with a slightly larger benefit for the *SpamC* code than for *LinReg*. The differences between the use of 1 and 8 threads are also remarkable, and analogous explanations to those of the IS and 3D FFT kernels in Section 3.6 apply here: the use of intensive InfiniBand communications by several threads in the same node limits the scalability of the applications when using more than 512 threads.

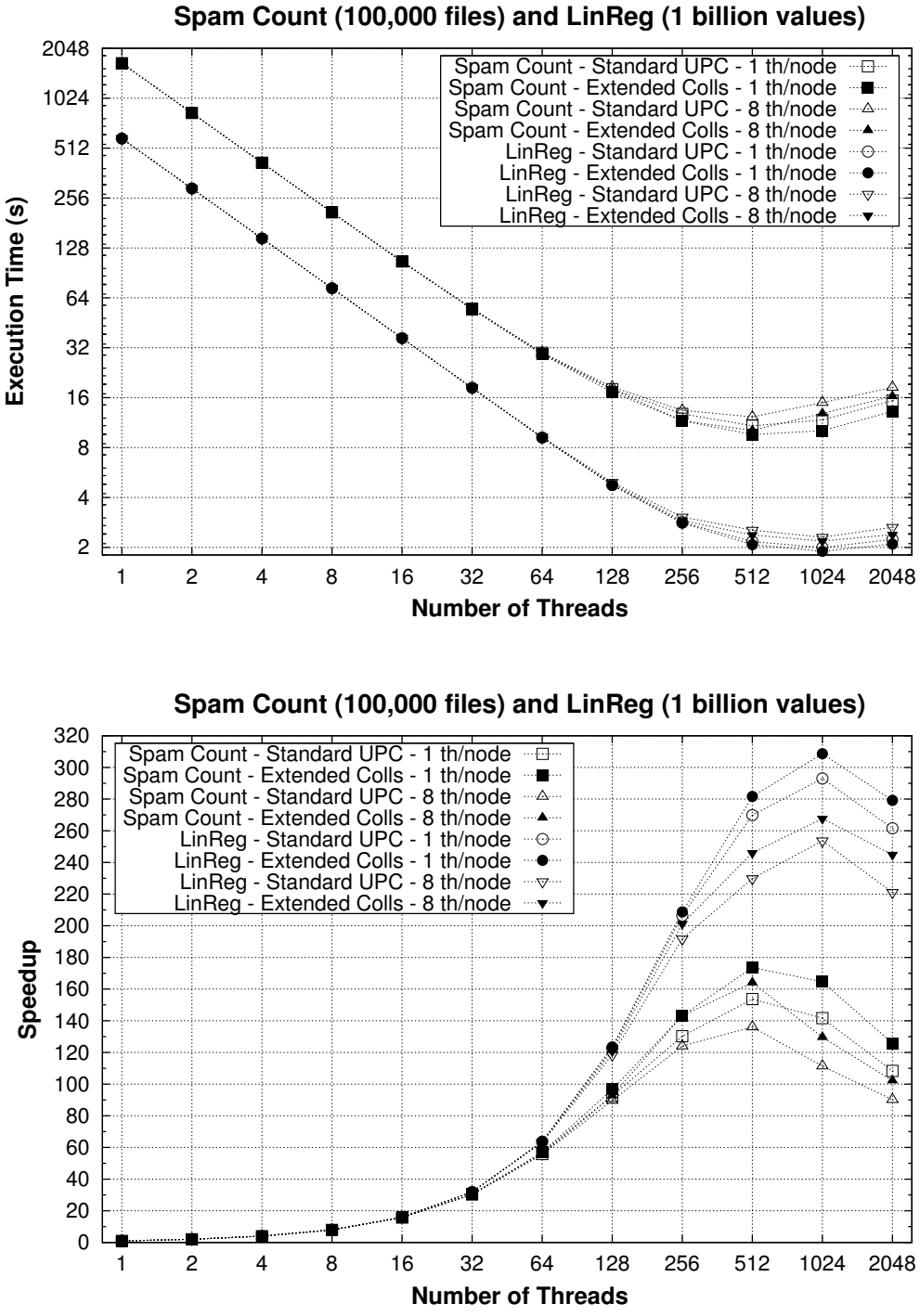


Fig. 4.5: Performance of *SpamC* and *LinReg* on hybrid shared/distributed memory (JRP)

4.4. Conclusions to Chapter 4

This chapter has presented the UPC-MR implementation, which has been developed to process MapReduce workloads taking advantage of the productive programming of the PGAS memory model. This framework consists of two management functions for each stage of processing (`ApplyMap` and `ApplyReduce`) that can manipulate any kind of input elements in a generic way, using two sequential user-defined *map* and *reduce* functions and allowing flexible control on the workload distribution at the "Map" stage, either automatically or user-defined. The goals of abstraction and usability are obtained by the generic definition of the functions, in order to be used as a template for any kind of input and output elements. However, all the developed codes have always sought good performance by using the necessary optimizations at low level, such as some extended collective functions for the design of communications at the "Reduce" stage.

UPC-MR has been evaluated on shared and distributed memory environments using four representative applications, comparing its performance with two MapReduce frameworks oriented to HPC workloads, Phoenix and MapReduce-MPI. According to the evaluations accomplished, UPC-MR has obtained similar or better results than those frameworks on all shared and distributed memory executions, because of the use of a generic and simple UPC framework that is based only on its own language constructs, without calling external libraries, and also relying on the efficient extended collectives library.

As a result, UPC has proved to be a good and feasible solution for implementing a MapReduce framework in order to execute codes on different HPC environments, offering programmability without performance penalties, even being able to achieve performance advantages.

Chapter 5

Parallel Simulation of Brownian Dynamics

This chapter presents the parallelization of a computationally-intensive scientific application in order to assess the programmability of UPC compared to alternative parallel approaches (MPI and OpenMP) and the benefits of the use of the extended collectives library, while also characterizing its impact on performance. The selected code is the simulation of particles in a fluid based on Brownian dynamics. This type of simulations are a very useful tool in many areas of biology and physics in order to determine the evolution of a system over the time. The target systems in Brownian dynamics are defined as a set of particles in a fluid, in which the simulation calculates the movement of the particles in a period of time by considering the interactions between particles and the interaction of particles with the surrounding medium, which are described by statistical properties. The calculation of interactions is a highly parallelizable process, but there are several dependencies in the system that have to be considered to obtain a correct simulation. As a result, the workload distribution and the achievement of high scalability from these simulations represent a challenging problem.

In the following sections, the Brownian dynamics simulation will be described and studied in computational terms taking an optimized sequential C code as reference. After defining the basis of the simulation, different possibilities of parallelization will be analyzed in terms of programmability and performance on shared and distributed

memory using three approaches: UPC, MPI and OpenMP. Finally, the benefits and drawbacks of every implementation in different testbed systems will be evaluated, and the lessons learned from this study will be commented.

5.1. Introduction to the Simulation of Brownian Dynamics

Particle-based simulation methods have been continuously used in physics and biology to model the behavior of different elements (e.g., molecules, cells) in a medium (e.g., fluid, gas) under thermodynamical conditions (e.g., temperature, density). These methods represent a simplification of the real-world scenario but often provide enough details to model and predict the state and evolution of a system on a given time and length scale. At this point, Brownian dynamics simulations describe the movement of particles in a solvent medium in a determined period of time, using several assumptions: for example, the size of the simulated particles is considered as significantly larger than the particles that make up the fluid, so that their individual interactions can be reduced to statistical fluctuations, which are modeled using a mobility tensor that contains information about the velocity field in the system. More technically, a finite difference scheme is applied to calculate the trajectory for each particle as a succession of short displacements Δt in time. In a system, containing N particles, the trajectory $\{\mathbf{r}_i(t); t \in [0, t_{max}]\}$ of particle i is calculated as a succession of small and fixed time step increments Δt. The time step is selected thereby: (1) large enough with $\Delta t \gg m_i/6\pi\eta a_i$, with η the solvent viscosity, a_i the radius and m_i the mass of the solute particle i, so that the interaction between individual fluid particles and the solutes can be averaged and coupled to the solutes via the diffusion tensor; and (2) small enough so that the forces and gradients of the diffusion tensor can be considered constant within Δt. According to these conditions, the simulation can be performed by calculating the forces that act on every particle in a time step, determining new positions for all particles and continuing this process in the following time step.

Brownian dynamics simulations are nowadays used to perform many studies in different areas of physics and biology [103], and there are several software tools

that help implementing these simulations, such as BrownDye [37] and the BROWNFLEX program included in the SIMUFLEX suite [30]. Some relevant work has also been published on parallel implementations of these simulations on GPUs [14], also including a simulation suite called BD_BOX [17]. However, there is still little information on parallelization methods for these simulations, especially about their performance and scalability in high performance computer systems.

The next sections provide an accurate description of the parallelization of a Brownian dynamics simulation in order to build a suitable implementation for HPC using different approaches. Section 5.2 presents the formal description of the simulation, and then Section 5.3 contains a more detailed description of its code implementation. In Section 5.4 details about the parallelization are outlined and Section 5.5 presents performance results of the parallel codes using different workloads and computational resources. Finally, Section 5.6 extracts the main conclusions from this work.

5.2. Theoretical Basis of Brownian Dynamics

The equation of motion, governing Brownian dynamics for N solvated molecules in a fluid, has been stated by Ermak and McCammon [23] (based on the Fokker-Planck and Langevin descriptions):

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \sum_{j=1}^{N} \frac{\partial \mathbf{D}_{ij}(t)}{\partial \mathbf{r}_j}\Delta t + \sum_{j=1}^{N} \frac{1}{k_B T}\mathbf{D}_{ij}(t)\mathbf{F}_j(t)\Delta t + \mathbf{R}_i(t+\Delta t) \quad (5.1)$$

This one-step propagation scheme takes into account the coupling of the particles to the flow field via the diffusion tensor $\mathbf{D} \in \mathbb{R}^{3N\times 3N}$ and the systematic forces $\mathbf{F}$, acting onto the particles with the global property $\sum_j \mathbf{F}_j = 0$. The vector $\mathbf{R} \in \mathbb{R}^{3N}$ contains correlated Gaussian random numbers with zero mean, which are constructed according to the fluctuation-dissipation theorem, with $\mathbf{R}_i \in \mathbb{R}^3$ and $\mathbf{D}_{ij} \in \mathbb{R}^{3\times 3}$, being sub-vectors and block-matrices corresponding to particle i and particle pairs i, j, respectively. $k_B T$ is the thermal energy of the system, where T is the temperature and k_B the Boltzmann constant.

The systematic interactions between particles are modeled by a Lennard-Jones type potential, from which the forces are obtained via the negative gradient:

$$V(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{5.2a}$$

$$\mathbf{F}_{ij} = -\nabla_{\mathbf{r}_{ij}} V(r_{ij}) = 24\varepsilon \left[2 \left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \frac{\hat{\mathbf{r}}_{ij}}{r_{ij}^2} \tag{5.2b}$$

where σ is the diameter of the particles and ϵ is the depth of the potential minimum. This potential has a short range character and practically interactions between particles are neglected for mutual distances $r_{ij} > R_c$, where R_c is the radius of a so called cutoff sphere, which is chosen as $R_c = 2.5\,\sigma$. The distance r_{ij} is chosen according to the minimum image convention, i.e. the shortest distance between particle i (located in the central simulation box) and particle j or one of its periodic images is taken into account (see Figure 5.1). In the code, the diffusion tensor $\mathbf{D}$ is calculated in periodic images, which implies a summation of particle pair contributions over all periodic images.

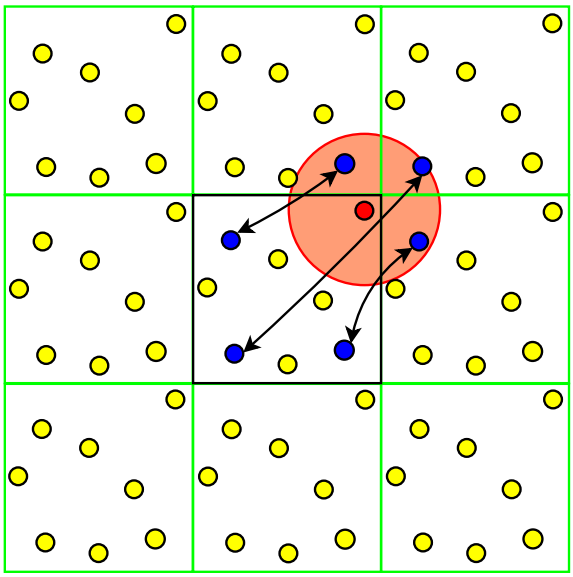


Fig. 5.1: Example of the short range interaction model with periodic boundary conditions

Depending on the approximation for the diffusion tensor, the partial derivative on the right-hand side of Eq. 5.1 might drop out. Thus, to obtain this result, the

regularized version of the Rotne-Prager tensor [67, 72] is selected. Applying this form of the diffusion tensor, the displacement vector of the Brownian particles, $\Delta\mathbf{r} = \mathbf{r}(t + \Delta t) - \mathbf{r}(t)$, can be rewritten in a more simple way:

$$\Delta\mathbf{r} = \frac{1}{kT}\mathbf{D}\mathbf{F}\Delta t + \sqrt{2\Delta t}\,\mathbf{Z}\boldsymbol{\xi} \tag{5.3}$$

where $\boldsymbol{\xi}$ is a vector of independent Gaussian random numbers. According to the previous simplifications, $\mathbf{Z}$ may be calculated via a Cholesky decomposition or via the square root of $\mathbf{D}$. Both approaches are very CPU-time consuming with a computational complexity of $\mathcal{O}(N^3)$ and impose a large computational load. Therefore the development of faster and more efficient and scalable methods with smaller complexity is an important task, and here Fixman [26] applied an expansion of the random displacement vector $\mathbf{R}$ in terms of Chebyshev polynomials, approximating its values without constructing $\mathbf{Z}$ explicitly and reducing the computational complexity to $\mathcal{O}(N^{2.25})$. Both methods for the construction of correlated random variates, based on the Cholesky decomposition and the Chebyshev approximation, will be considered for the present simulation code.

5.3. Implementation of the Simulation Code

The parallelization of the Brownian dynamics simulation has taken an existing optimized sequential C code as basis. The system under study consists of a cubic box where periodic boundary conditions are applied, and the propagation of Brownian particles is performed by evaluating Eq. 5.3. The main component of the code is a `for` loop. Each iteration of this loop corresponds to a time step in the simulation, which calls several functions, being `calc_force()` and `covar()` the most time consuming and thus the main targets of the parallelization. Function `calc_force()` includes: (1) the propagation of particles (with $\mathcal{O}(N)$ complexity, where N is the number of particles in the system), (2) the force computation, for which a linked-cell technique [2] is used ($\mathcal{O}(N)$), and (3) the setup of the diffusion tensor ($\mathcal{O}(N^2)$). The correlated random displacements are calculated in function `covar()` via a Cholesky decomposition ($\mathcal{O}(N^3)$) and alternatively via the Fixman's approximation method, with complexity $\mathcal{O}(N^{2.25})$ [26].

The values of the diffusion tensor **D** are computed in `calc_force()` for every pair of particles in the system according to the distance between them in every dimension following the Rotne-Prager tensor description, and then stored using double precision floating point values in matrix `D`, which is declared as a square `pNDIM`×`pNDIM` matrix, where `pNDIM` is the product of the total number of particles (N) and the system dimensions (3). Thus, each row/column in `D` is associated to a particle and a dimension (for example, value `D[3*4+2][3*8+2]` contains the diffusion tensor between particles 4 and 8 in the third dimension z). The interaction values in `D[a][b]` are also stored in `D[b][a]`, being `D` symmetric, i.e. the values of the upper triangular part of `D` are computed and they are copied to the corresponding positions in the lower triangular part. This data replication has several advantages for the sequential algorithm because it helps simplifying the implementation of the operations with matrix `D`, allowing more straightforward computations on this matrix.

After the initialization of `D` in `calc_force()`, function `covar()` calculates the random displacement vector **R**, which requires the construction of correlated Gaussian random numbers with given covariance values. As stated in Section 5.2 (see Eq. 5.3), the random vector is written as:

$$\mathbf{R} = \sqrt{2\Delta t}\,\mathbf{Z}\boldsymbol{\xi} \tag{5.4}$$

where the random values $\boldsymbol{\xi}$ are obtained using the pseudo-random generator `rand()`, and **Z** can be obtained either as a lower triangular matrix **L** from a Cholesky decomposition or as the matrix square root **S**. To obtain the entries of matrix **L**, the following procedure is applied:

$$L_{ii} = \sqrt{D_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}\ \text{, for diagonal values in } \mathbf{L} \tag{5.5a}$$

$$L_{ij} = \frac{1}{L_{jj}}\left(D_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}\right)\text{, where i} > \text{j} \tag{5.5b}$$

$$R_i = \sqrt{2\Delta t}\sum_{j=1}^{i} L_{ij}\,\xi_j \tag{5.5c}$$

i.e. having calculated the diffusion tensor matrix `D`, a Cholesky decomposition generates the lower triangular matrix `L`, from where $\mathbf{R}$ is generated via a matrix-vector product between `L` and $\boldsymbol{\xi}$.

The alternative to this method is the use of Fixman's algorithm, which implements a fast procedure to approximate the matrix square root $\mathbf{S}$ using Chebyshev polynomial series. The degree of the polynomial is fixed depending on the desired accuracy for the computed values. The advantage of this method is that it does not need to build the matrix $\mathbf{S}$ explicitly, but constructs directly an approximation to the product $\mathbf{S}\boldsymbol{\xi}$ in an iterative way. The computations performed here represent a generalization of known series to obtain the scalar square root to the case of vectors:

$$\mathbf{S}_M = \sum_{m=0}^{M} a_m \mathbf{C}_m \tag{5.6a}$$

where M is the degree of the polynomial, which controls the accuracy of the approximation. In the limit it holds that $\lim_{M\to\infty} \mathbf{S}_M = \mathbf{S}$. Accordingly, the vector of correlated Gaussian variates becomes:

$$\mathbf{Z}_M = \mathbf{S}_M\boldsymbol{\xi} = \sum_{m=0}^{M} a_m \mathbf{C}_m\boldsymbol{\xi} = \sum_{m=0}^{M} a_m \mathbf{z}_m \tag{5.6b}$$

The coefficients a_m can be pre-computed [7] and the vectors $\mathbf{z}_m$ are computed within an iterative procedure:

$$\mathbf{z}_0 = \boldsymbol{\xi} \;\;;\;\; \mathbf{z}_1 = (d_a\mathbf{D} + d_b\mathbf{I})\boldsymbol{\xi} \;\;;\;\; \mathbf{z}_{m+1} = 2\,(d_a\mathbf{D} + d_b\mathbf{I})\mathbf{z}_m - \mathbf{z}_{m-1} \tag{5.7a}$$

with

$$d_a = \frac{2}{\lambda_{max} - \lambda_{min}} \;\;;\;\; d_b = \frac{\lambda_{max} + \lambda_{min}}{\lambda_{max} - \lambda_{min}} \tag{5.7b}$$

where λ_{min}, λ_{max} are the upper and lower limits for the eigenvalues of $\mathbf{D}$ [40, 73].

The approximation vectors $\mathbf{z}_m$, as well as the maximum and minimum eigenvalues of $\mathbf{D}$, are computed analogously using two separate iterative algorithms. These algorithms apply the technique of double buffering, i.e. use a pair of arrays that are read or written alternatively on each iteration: every approximation (associated to a particle in a dimension) is computed using all the approximated values from the

previous iteration, therefore the use of two arrays avoids additional data copies and data overwriting. This procedure will be illustrated in Section 5.4.4.

After computing the random displacements in an iteration, the function `move()` performs the matrix-vector product between the diffusion matrix and the force vector $\mathbf{F}$, and adds all the computed contributions to obtain the new positions of the particles (reduction operation). The matrix-vector product has $\mathcal{O}(N^2)$ complexity for this function. Some additional physical values (e.g., pressure) are also computed here to monitor and control the progress of the simulation.

Table 5.1 presents the breakdown of the execution time of the sequential program using a node of the JuRoPa supercomputer (which is also used in Section 5.5 for the performance evaluation) in terms of the previously discussed functions, using 256 and 1024 input particles for 50 time steps of simulation. The diffusion tensor matrix `D` has $(3 \times N)^2$ elements, thus its construction takes at least a complexity of $\mathcal{O}(N^2)$. This is true for the real space contributions of the Ewald sum as the cutoff radius is of the order of half the system size (or even larger), in order to keep the reciprocal space contribution, i.e. the number of k-values, small for a given error tolerance. Since the mutual distances between particles are calculated in the real space contribution, it is natural to integrate the construction of matrix `D` in the calculation of short range direct interactions between particles (whose complexity is $\mathcal{O}(N)$), thus giving out the $\mathcal{O}(N^2)$ complexity stated in row "short range contributions" of Table 5.1.

Code Part	**Complexity**	**N = 256**	**N = 1024**
`calc_force()` - short range contributions	$\mathcal{O}(N^2)$	4.733 s	75.966 s
`calc_force()` - long range contributions	$\mathcal{O}(N^{2.5})$	7.095 s	181.103 s
`covar()` - option 1: Cholesky	$\mathcal{O}(N^3)$	3.733 s	250.578 s
`covar()` - option 2: Fixman	$\mathcal{O}(N^{2.25})$	0.762 s	17.735 s
`move()`	$\mathcal{O}(N^2)$	0.019 s	0.341 s
Total time (with Cholesky)	$\mathcal{O}(N^3)$	15.580 s	507.988 s
Total time (with Fixman)	$\mathcal{O}(N^{2.5})$	12.609 s	275.145 s

Table 5.1: Breakdown of the execution time of the sequential code

The long range contribution to the diffusion tensor also has to be calculated for every matrix element, i.e. for each particle pair, which also imposes a computational complexity of $\mathcal{O}(N^2)$. However, there is an additional contribution to the long range part, giving rise to a larger complexity, since a set of reciprocal vectors has to be

considered to fulfill a prescribed error tolerance in the Ewald sum, increasing the complexity to approximately $\mathcal{O}(N^{2.5})$. The execution times of `covar()` also tend to reveal the complexities of the algorithms, commented in Section 5.2.

5.4. Development of the Parallel Implementation

The exponential increase of the total execution time with the number of particles is the main motivation for the development of a parallel implementation of this code. Three different approaches have been used: (1) a directive-based approach for shared memory with OpenMP, (2) a message-passing programming model with the MPI C library, and (3) the PGAS model using UPC. OpenMP and MPI represent traditional and widely used approaches for parallel programming on shared and distributed memory, whereas UPC focuses on providing an efficient approach for hybrid shared/distributed memory environments. Here all these approaches are analyzed in different environments, in order to obtain the best solutions for them.

5.4.1. Parallel Workload Decomposition

A preliminary analysis of the structure of the simulation code reveals that each iteration of the main loop, i.e. each time step, has a strong dependency on the previous iteration, because the position of a particle depends on the computations of previous time steps. Therefore, these iterations cannot be executed concurrently, so the work distribution is only possible within each iteration. At this point, the main parallelization efforts have to be focused on the workload decomposition of `calc_force()`, according to Table 5.1, but also considering the performance bottlenecks that might arise when performing communications, especially in `covar()`.

In order to facilitate the parallel computations needed to update the diffusion tensor values and random displacements, all processes require to have access to the coordinates for every particle in the system (for simplicity, the term “processes” will be used to denote both processes in MPI and threads in UPC or OpenMP). Thus, all processes store all the initial coordinates of the particles to avoid continuous remote calls to obtain the necessary coordinate values. After each iteration, all coordinates

are updated for every process by means of function `move()`, thus minimizing communications. Moreover, this assumption allows the parallel computation of short and long range contributions without communications in MPI and UPC: each process can compute any element of matrix `D` independently from the rest, and therefore the parallelization of `calc_force()` becomes straightforward. The tradeoff of this approach in MPI and UPC is a slightly higher memory consumption (approximately `pNDIM` double precision additional values), whereas the OpenMP code is not affected: the parallel regions are only defined after the initial coordinates are read.

The calculation of each random displacement in `covar()` depends on many of the elements of matrix `D`, whose computation has been previously performed by different processes in `calc_force()`, and consequently communications are unavoidable here. Therefore, it is necessary to find a suitable workload distribution of diffusion tensor values in matrix `D` to favor the scalability of the code by minimizing the amount of communications required by `covar()`. The following subsections present different approaches to increase the scalability of the code taking advantage of the specific features of OpenMP, MPI and UPC.

5.4.2. Shared Memory Parallelization (UPC & OpenMP)

The use of the shared memory space in UPC to store matrix `D` allows a straightforward shared memory parallelization which is comparable in terms of programmability with OpenMP. Figure 5.2 shows the distribution of `D` for 4 threads using a sample matrix, divided in four chunks of equal size (one per thread). Each element on it represents the diffusion tensor values associated to a pair of particles in every combination of their dimensions, that is, a 3×3 submatrix. In UPC, all threads are able to access all the data stored in the shared memory region, so this parallelization only requires changes in the matrix indexing to support the access in parallel by UPC threads. Here, the matrix is distributed in blocks of $N/THREADS$ elements (i.e., 3×3 submatrices) among threads. Each diffusion tensor value from the upper triangular part of the matrix is computed by the thread to which it has affinity (see left graph). After that, the upper triangular part is copied into the lower triangular part (see right graph), as described in Section 5.3, using implicit one-sided transfers initiated by the thread that has the source data by means of assignments.

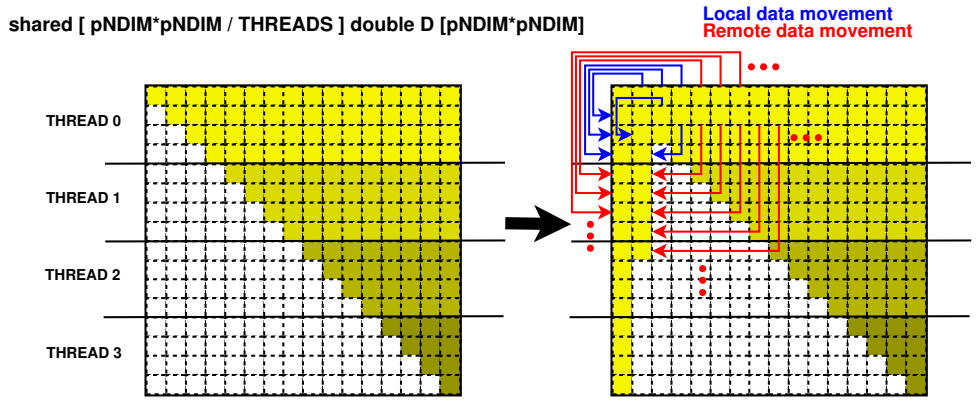


Fig. 5.2: Work and data distribution with `D` as a shared matrix in UPC

This approach has allowed the quick implementation of a prototype of the parallel code, providing a straightforward solution also for distributed memory architectures thanks to the shared memory view provided by UPC. This represents a significant advantage over MPI, where the development of an equivalent version of this parallelization on distributed memory would require a significantly higher programming effort because of the lack of a shared memory view (data have to be transferred explicitly among processes), showing poorer productivity.

However, there are two drawbacks in this parallelization. The first one is its poor load balancing: thread 0 performs much more work than the last thread (*THREADS*-1). This workload imbalance can be partly alleviated through the distribution of rows in a cyclic way, but this will not be enough when the number of threads is large and few rows are assigned to each thread. The second issue of this work distribution is the inefficiency of single-valued remote memory copies [22], which is due to the use of virtual memory addresses to map the shared variables in UPC. While private variables have addresses that are directly translated to physical positions in memory, shared variables use an identifier that defines the affinity of the variable, the data block to which it belongs and the offset of the variable inside the data block (see Section 2.1.1). As a result, the computational cost of handling these shared address translations is not acceptable when simulating large systems for a long period of time.

Regarding the OpenMP code, the simulation is parallelized only by inserting directives in the code, without further changes. However, the original sequential

algorithm for Cholesky decomposition of matrix `D` calculates the first three values of the lower triangular result matrix (that is, the elements of the first two rows), and then the remaining values are computed row by row in a loop. Here the data dependencies between the values in `L` computed in each iteration and the ones computed in previous iterations prevent the direct parallelization of the loop. Thus, an alternative iterative algorithm is proposed: the first column of the result matrix is calculated by dividing the values of the first row in the source matrix by its diagonal value (parallelizable `for` loop), then these values are used to compute a partial contribution to the rest of elements in the matrix (also parallelizable `for` loop). Finally, these two steps are repeated to obtain the rest of columns in the result matrix. Listing 5.1 shows the source code of the iterative algorithm: a `static` scheduling is used in the first two loops, whereas the last two use a `dynamic` one because of the different workload associated to their iterations.

```
 L[0][0] = sqrt(D[0][0]);
#pragma omp parallel private(i,j,k)
{
#pragma omp for schedule(static)
 for (j=1; j<N*DIM; j++)
   L[j][0] = D[0][j]/L[0][0];
#pragma omp for schedule(static)
 for (j=1; j<N*DIM; j++)
   for (k=1; k<=j; k++)
     L[j][k] = D[k][j]-L[0][j]*L[0][k];

 for (i=1; i<N*DIM; i++) {
   L[i][i] = sqrt(L[i][i]);
#pragma omp for schedule(dynamic)
   for (j=i+1; j<N*DIM; j++)
     L[j][i] = L[i][j]/L[i][i];

#pragma omp for schedule(dynamic)
   for (j=i+1; j<N*DIM; j++)
     for (k=i+1; k<=j; k++)
       L[j][k] -= L[i][j]*L[i][k];
 }
}
```

List. 5.1: Iterative Cholesky decomposition with OpenMP

Two iterative approximation methods are used for Fixman's algorithm: (1) the calculation of the minimum and maximum eigenvalues of matrix `D`, which uses a variant of the power method, and (2) the computation of coefficients following the

formulae in Eq. 5.7. In both cases, the core of the algorithm consists of two `for` loops that operate on every row of matrix `D` independently (thus fully parallelizable) until the predefined accuracy is reached. Each iteration of these methods requires all the approximations computed for every particle in all dimensions (`pNDIM` values) in the previous iteration, thus both codes use two arrays that are read or written alternatively on each iteration to avoid extra data copies, as mentioned in Section 5.3.

When using Fixman's algorithm, the OpenMP code includes `omp for` directives for each loop to parallelize all iterative approximation methods, using a `critical` directive to obtain the maximum/minimum eigenvalues of `D` and a `reduction` clause to compute the total error value for the final approximated coefficients. Listing 5.2 presents the pseudocode of the loop that computes the maximum eigenvalue of `D`, using `x` and `x_d` as working arrays for the iterative method. Here a `reduction` clause with a maximum operator could be used instead of the `critical` directive, but in this case the experimental analysis showed better performance using `critical`.

```
#pragma omp parallel
{
while the required accuracy in 'eigmax' is not reached
do
  if the iteration number is even
    // read from 'x', write to 'x_d'
#pragma omp for schedule(static) \
    default(shared) private(i,j,...)
    for every particle 'i' in the system
      x_d[i] = 0;
      for every particle 'j' in the system
        x_d[i] += D[i][j]*x[j]/eigmax;
      endfor
#pragma omp critical
      if it is the maximum value
        select it in 'eigmax'
      endif
    endfor
  else
    // The same code as above, but swapping 'x' and 'x_d'
  endif
endwhile
}
```

List. 5.2: OpenMP pseudocode that computes the maximum eigenvalue of `D` in Fixman's algorithm

The efficiency of the OpenMP algorithms is significantly better than the commented UPC approach, even though the differences in programmability are not significant. However, in both cases the implemented codes cannot obtain reasonable performance when distributed memory communications are involved. The shared address translations and the implicit remote data movements are not able to provide scalability for internode communications, because of the high number of data dependencies for the calculation of random displacements. Therefore, a different approach is required for executions on more than one node.

5.4.3. Distributed Memory Parallelization (UPC & MPI)

Figure 5.3 presents the distribution of matrix `D` and its associated data movements for a balanced workload decomposition on private distributed memory, where each process is assigned a set of particles in the system in order to compute their corresponding diffusion tensor values. Here the number of particles associated to a process is not evenly divided, but instead follows an analogous approach to the force-stripped row decomposition scheme proposed by Murty and Okunbor [55], with the goal of achieving a more balanced number of computations and remote copies in matrix `D`. This workload/domain decomposition consists in distributing the number of elements in the upper triangular part of matrix `D` (`pNDIM`$\times$`(pNDIM+1)/2`, defined as `nelems` in the code) between the number of processes in the program by assigning consecutive rows to process i until the total number of assigned diffusion tensor values is equal to or higher than `nlocalelems*(i+1)`, where `nlocalelems` is `nelems` divided by the number of processes.

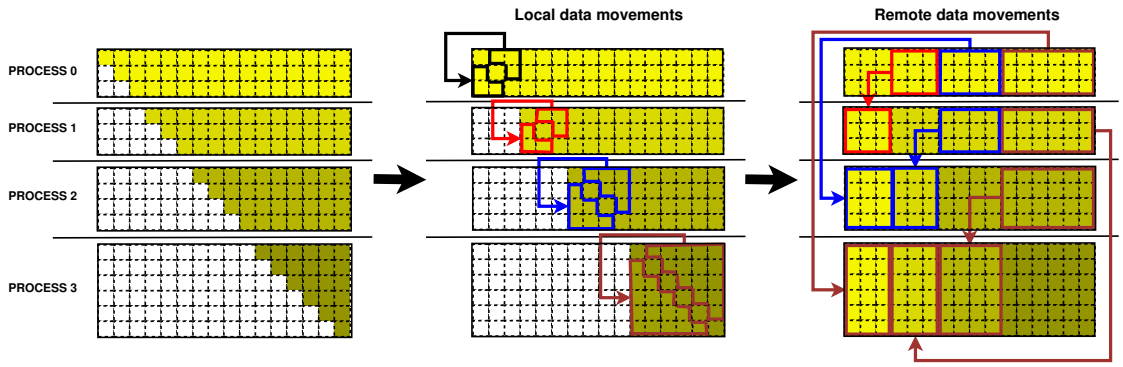


Fig. 5.3: Balanced workload decomposition of `D` on distributed memory

This approach has been implemented both in MPI and UPC. In both scenarios, first all computations of diffusion tensor values are performed locally by each process, and then the values are moved to the corresponding destination in the lower triangular part of matrix `D`. Sometimes the destination position is local to the process which has the source data (see local data movements in the middle graph of Figure 5.3), whereas on the remaining cases there are data transfers between processes (see remote data movements in the rightmost graph of Figure 5.3). Regarding the UPC parallelization, matrix `D` has been defined as a private memory region (local to each process) for each submatrix. The reason for this decision is that the UPC standard does not allow the allocation of a different number of elements per process in the shared memory region, i.e. the block size of a shared array must be the same for all processes.

Despite the relatively good balancing of this distribution, its main drawback is the significant overhead associated to the communications needed to achieve the symmetry in matrix `D`. After obtaining the symmetric matrix, the next step in the simulation (function `covar()`) involves either a Cholesky decomposition or a Fixman approximation. The Cholesky decomposition can take advantage of the previous matrix distribution, minimizing the number of remote communications. Regarding Fixman's algorithm, it is really convenient to fill the lower triangular part of matrix `D` in order to avoid smaller element-by-element data movements in `covar()`, but the communication time may become too high even when few processes are used. This is also confirmed by the results of Table 5.1: the sequential computation of the interactions for 1024 particles takes less than 5 minutes for 50 time steps, being the average calculation time of each tensor value of about 0.6 microseconds in each iteration, and after that a large percentage of these computed values is remotely copied (about 68% in the example of Figure 5.3). As a result, the cost of communications can easily represent a significant percentage of the total execution time.

5.4.4. Optimizations for Distributed Memory

An analysis of the previously presented parallel implementations has shown different matrix decomposition options for `D`, as well as their associated drawbacks, which have a significant impact on the random displacement generation method

considered in `covar()` (Cholesky or Fixman). Thus, the optimization of these implementations has taken into account different factors depending on the random displacement algorithm, especially for Fixman's method, in order to exploit data locality and achieve higher performance with MPI and UPC. The next subsections present the optimized parallel algorithms for the computation of random displacements in `covar()`.

Optimized Parallel Cholesky Decomposition

The optimized parallel Cholesky decomposition is based on the balanced distribution presented in Section 5.4.3 (see left graph of Figure 5.3), and minimizes communications by introducing some changes with respect to the sequential algorithm; more specifically, this parallel code does not fill the lower triangular part of `D`, and performs an efficient workload partitioning that maximizes data locality. Listing 5.3 presents the pseudocode of the algorithm. Initially, process 0 computes the first column of the result matrix `L` using its first row in `D`, calculates the random displacement associated to this first row, and then broadcasts to the other processes the computed values of `L`, that are stored in an auxiliary array of `pNDIM` elements (`L_row`). In the UPC code, as all variables are private, the broadcast function used here is the *priv* variant of the broadcast collective (whose behavior was illustrated in Figure 3.7). Once all processes have the auxiliary array values, two partial calculations are performed in parallel: (1) a contribution to obtain the elements of matrix `L` in the positions of their assigned elements of matrix `D`, and (2) a contribution to the random displacement associated to each of their rows. These steps are also applied to the rest of rows in matrix `D`.

This algorithm reduces the data dependencies by breaking down the computations of the elements in `L` and the final random displacements: both sets of values are computed as sums of product terms (cf. Listing 5.3 and Eq. 5.5 in Section 5.3), thus a process can compute these sums partially as soon as it obtains one of their associated terms. As a result, the random displacements of every particle in every dimension are calculated step by step, showing a fine-grain workload decomposition that maximizes the scalability of the code. Moreover, this algorithm takes advantage of a distribution of matrix `L` between processes similar to that of matrix `D`, where each process builds only the submatrix associated to its particles, and the

only additional storage required is the `L_row` array that receives the values of the processed row in each iteration.

```
for every row 'k' in matrix D
  if the row is local to this process
    if it is the first row in D (row 0)
      L[0][0] = sqrt(D[0][0]);
      for every element 'i' in this row
        L[0][i] = D[0][i]/L[0][0];
      endfor
    else
      L[k][k] = sqrt(L[k][k]);
      for every element 'i' in this row
        L[k][i] = L[k][i]/L[k][k];
      endfor
    endif
    displacement[k] += L[k][k]*random_displ[k];
  endif

  broadcast values L[k][k:pNDIM-1] to 'L_row'

  for every row 'j' > 'k' in matrix D
    if the row is local to this process
      for every element 'i' >= 'j' in this row
        if it is the first row in D (row 0)
          L[j][i] = D[j][i] - L_row[j]*L_row[i];
        else
          L[j][i] -= L_row[j-k]*L_row[i-k];
        endif
      endfor
      displacement[j] += L_row[j-k]*random_displ[k];
    endif
  endfor
endfor
```

List. 5.3: Pseudocode for the computation of displacements with Cholesky decomposition (distributed memory)

Fixman's Algorithm with Balanced Communications

Figure 5.4 presents a domain decomposition and hence a workload distribution, both for MPI and UPC, focused on maximizing load balancing and exploiting local computations to reduce communications for Fixman's algorithm. Matrix `D` consists of diagonal and non-diagonal elements (where each element is a 3×3 submatrix that

contains the interaction values for two particles for each combination of its dimensions), and the distribution of `D` assigns to each process a balanced number of consecutive elements of each type, regardless of the particles to which they correspond. Thus, in Figure 5.4 the 16 diagonal elements are distributed among the 4 processes (each one receives 4 diagonal elements), and the 120 remaining elements are scattered (30 elements per process). Finally, every chunk is linearized in `arrayDiag` (diagonal chunks) and `arrayD` (non-diagonal chunks) following the flattening process shown at the bottom of the figure for every 3×3 submatrix. This distribution favors local processing for diagonal values, as well as the balanced distribution of data and communications for non-diagonal values.

Listing 5.4 presents the pseudocode that implements the parallel computation of the maximum eigenvalue in matrix `D` through an iterative method, which can be analogously applied to the subsequent approximation of the minimum eigenvalue and the elements of matrix **S**, as commented for the sequential code in Section 5.3. Each process calculates locally a partial result for the approximated eigenvalue using its assigned diffusion tensor values, and then an all-to-all collective communication (in UPC, the *priv* in-place exchange from the extended collectives library) is invoked by every process to get all the partial results of its assigned rows. Finally, each process computes the total approximation of its associated eigenvalues, an allgather collective operation (in UPC, the *priv* in-place allgather from the extended collectives) is used to provide all processes with all the approximations, and an allreduce collective (in UPC, the extended *priv* allreduce version) obtains the maximum eigenvalue of all processes in order to start a new iteration of the method. The computed eigenvalues are here assigned alternatively to arrays `eigenx` and `eigenx_d` with the goal of avoiding unnecessary data copies because of the iterative method, as commented in Section 5.3. As the distribution of particles is initially balanced, as well as the amount of communications performed by each process, there is no relevant workload difference among the processes, regardless of the number of iterations.

Besides load balancing, another benefit of this distribution is the optimized memory storage of variables, because it only considers the minimum number of elements that are necessary for the simulation instead of using the whole square matrix. Additionally, this implementation requires the use of indexes that mark, for example, the starting position of the values associated to each particle in a dimension, which

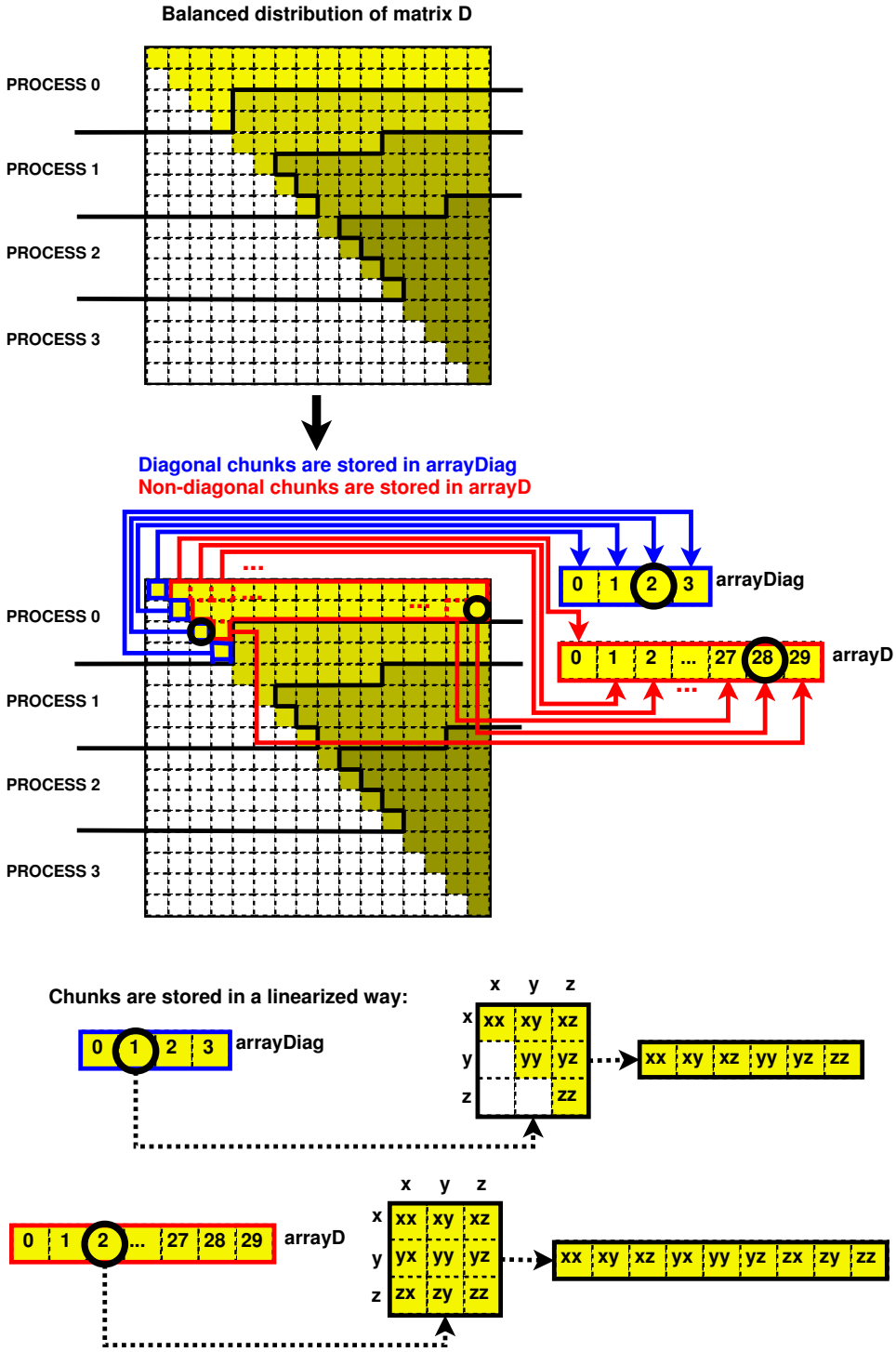


Fig. 5.4: Balanced distribution of matrix `D` for Fixman's algorithm (detailed data structures for process 0)

in turn increases slightly the complexity of the source code. However, these indexes are computed only once before the simulation loop begins, and therefore they do not have a significant influence on performance.

```
initialize maximum eigenvalue 'eigmax' to 1
initialize 'eigenx' (eigenvalue approximations per row) to 1
while the desired accuracy of 'eigmax' is not reached
  if the iteration number is even
    for every local element in 'arrayD'
      compute partial approximation of 'eigmax' \
       in 'Dcopy' using 'eigenx'
    endfor
    for every local element in 'arrayDiag'
      compute partial approximation directly \
       in 'eigenx_d' using 'eigenx'
    endfor
    all-to-all collective to get all partial
     approximations from 'Dcopy'
    for every local element 'i' in 'eigenx_d'
      for every process 'j'
        sum the partial result of process 'j' in 'Dcopy' \
         to get the final approximation of 'eigenx_d[i]'
      endfor
      if 'eigenx_d[i]' is the maximum value
        update 'eigmax'
      endif
    endfor
    allgather the computed values in 'eigenx_d'
    allreduce the maximum value of 'eigmax' from all processes
  else
    the same code as above, but swapping 'eigenx' and 'eigenx_d'
  endif
endwhile
```

List. 5.4: Pseudocode for the parallel computation of the maximum eigenvalue with Fixman's algorithm (distributed memory)

Fixman's Algorithm with Minimum Communications

The previous algorithm for Fixman's method has limited its scalability because of the overhead derived from the communications required at each iteration. In order to reduce the amount of communications, a block distribution of matrix `D` by rows is proposed in Figure 5.5, both for MPI and UPC. This distribution considers that the particles in the system are evenly distributed between processes, and each of

them computes all the diffusion tensor values of its associated particles. As a result, Fixman's algorithm can be implemented using a minimum number of communications: the approximations of the correlation coefficients for every particle in every dimension are always computed locally by the corresponding process, and only an extended *priv* allgather collective is necessary at the end of each iteration. The main drawback of this implementation is its higher computational cost, because it has to compute roughly double the number of elements, as it does not take full advantage of the symmetry of `D` (only locally to each process). However, the scalability of this approach is significantly higher than that of the previous algorithms, because of the reduced number of communications required, which allows to outperform the previous approaches as the number of processes increases.

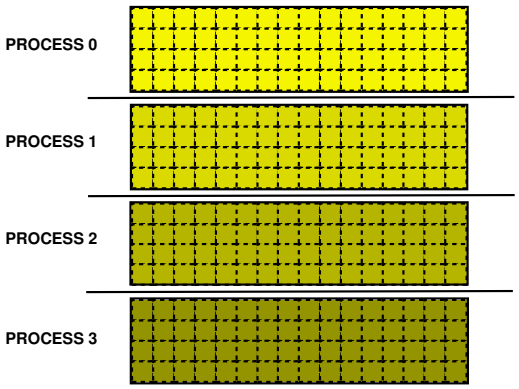


Figure 5.5: Work distribution with `D` as a full-size private matrix

5.5. Performance Evaluation

The evaluation of the developed parallel Brownian dynamics codes has been accomplished mainly on JRP, the JuRoPa supercomputer at Jülich Supercomputing Centre (see Section 3.5 for more details). Additionally, a second system has been used for shared memory executions: a node of the SVG supercomputer at CESGA [76], which is an HP ProLiant SL165z G7 node with 2 dodeca-core AMD Opteron processors 6174 (Magny-Cours) at 2.2 GHz with 32 GB of memory, and from now on is referred to as "MGC". The Intel C Compiler (icc) v12.1 and the Open64 Compiler Suite (opencc) v4.2.5.2 have been used as OpenMP compilers in JRP and MGC, respectively, with the environment variable `OMP_STACKSIZE` set

to a small value (128 KB) for OpenMP executions in order to obtain the highest efficiency. The UPC compiler used on both systems was Berkeley UPC v2.14.2 (released in May 2012) with the Intel C Compiler v12.1 as backend C compiler on both systems, and relying on the IBV conduit on JRP. ParaStation MPI 5.0.27-1 [61] has been used by the MPI code. All the executions in this evaluation were compiled with the optimization flag `-O3`, and periodic boundary conditions (3×3 boxes per dimension) were considered for all simulations.

In order to perform a fair comparison, all speedup results have been calculated taking the execution times of the original sequential C code as baseline, as it represents the fastest approach. The problem size considered for each graph (Figures 5.6-5.10) is fixed for a varying number of cores, thus showing strong scaling, and all tests on distributed memory use a fill-up policy for process scheduling in the nodes of the testbed system, i.e. always with one process per physical core. Different versions of the simulations are shown depending on the algorithm and work distribution used for the computation of random displacements: (1) using Cholesky decomposition (see Section 5.4.2 for OpenMP and Section 5.4.4 for MPI and UPC), (2) using Fixman's algorithm for OpenMP according to the algorithm in Section 5.4.2, (3) using Fixman's algorithm with the distribution presented in Section 5.4.4 that balances workload and communications for MPI and UPC (referred to as *bal-comms* from now on), and (4) using Fixman's algorithm with matrix `D` distributed to minimize communications for MPI and UPC, as described in Section 5.4.4 (referred to as *min-comms*). Figures 5.6 and 5.7 present the performance results of the simulations on shared memory for OpenMP and UPC, and Figures 5.8 to 5.10 present the results on distributed memory for MPI and UPC.

Figure 5.6 shows the execution times and speedups of the whole simulation with 256 particles and 100 time steps (double the workload used in Table 5.1). The execution times of JRP are better than those of MGC for both random displacement generation codes, even though each JRP node has only 8 physical cores which are able to run 16 simultaneous threads thanks to the use of Simultaneous Multithreading (SMT), but taking little advantage compared to the use of a single thread per core (8 threads per node). This is mainly due to the higher processor power of JRP.

The reduced problem size involves a low average workload per thread, thus the bottlenecks of the code (e.g., OpenMP `atomic` directives and UPC collective com-

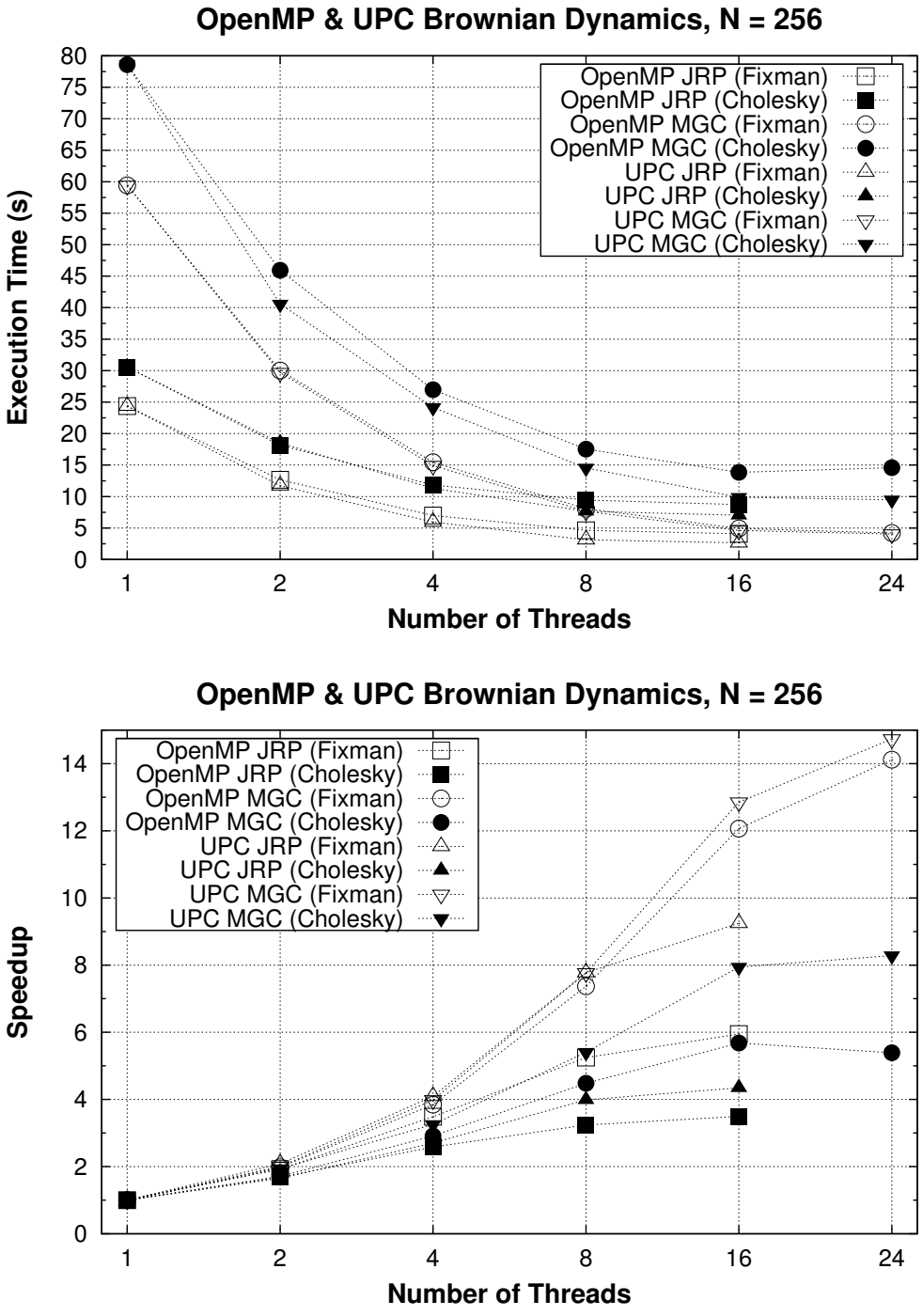


Figure 5.6: Shared memory performance results with 256 particles (Fixman and Cholesky)

munications) are more noticeable. Nevertheless, the UPC implementation even obtains superlinear speedup for 2 and 4 threads with Fixman on JRP. Regarding the `covar()` function, the use of Fixman's method clearly helps to obtain higher performance when compared to Cholesky decomposition, as stated in the previous sections.

Figure 5.7 shows the simulation results for a large number of particles (4096) and 50 time steps. Only the results of `covar()` with Fixman's algorithm are shown, because of its higher performance. The larger problem size provides higher speedups than those of Figure 5.6 for both systems. The algorithmic complexity of calculating the diffusion tensor $\mathbf{D}$ is $\mathcal{O}(N^2)$, whereas Fixman's algorithm is $\mathcal{O}(N^{2.25})$; thus, when the problem size increases, the generation of random displacements represents a larger percentage of the total simulation time. As a result of this, and also given the parallelization issues commented in Section 5.4.2, the speedup is slightly limited for 16 or more threads, mainly for OpenMP (also due to the use of SMT in JRP). However, considering the distance to the ideal speedup, both systems present reasonably good speedups for this code.

Figure 5.8 shows the execution times and speedups of the parallel simulation of 256 particles over 100 time steps (as in Figure 5.6) for distributed memory on JRP. The results show that, on the one hand, the best performance up to 16 cores is obtained by the version that uses Fixman with balanced workload and optimized storage (*bal-comms*), with very similar results for MPI and UPC. However, the weight of allgather and all-to-all communications in `covar()` limits heavily the scalability with this approach from 32 cores onward, as the ratio computation/communication time is low. On the other hand, Fixman *min-comms* presents the opposite situation: the redundant computations cause poor performance for a small number of cores, but the minimization of communications provides good scalability as the number of cores increases, even for 128 cores (that is, when only 2 particles per core are processed). The codes based on Cholesky are able to scale up to 32 cores, but performance decreases for 64 and 128 cores because of the significant weight of the communication overhead in the total execution time. These results also show that codes relying on Fixman's algorithm outperform Cholesky-based codes, either using the *bal-comms* version (except for a large number of cores) or the *min-comms* one. Additionally, the execution times when using Cholesky show a higher increase with

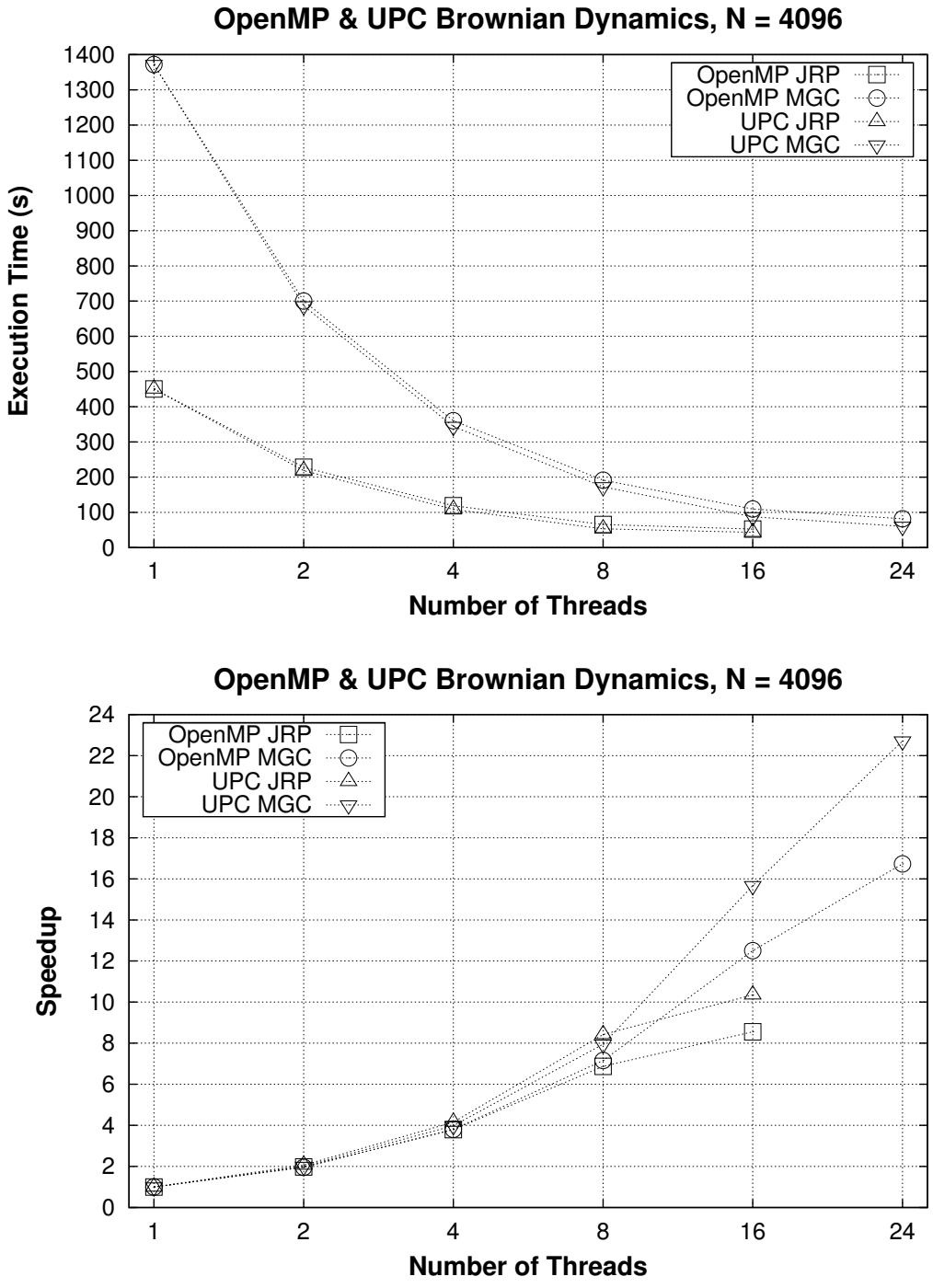


Figure 5.7: Shared memory performance results with 4096 particles (Fixman)

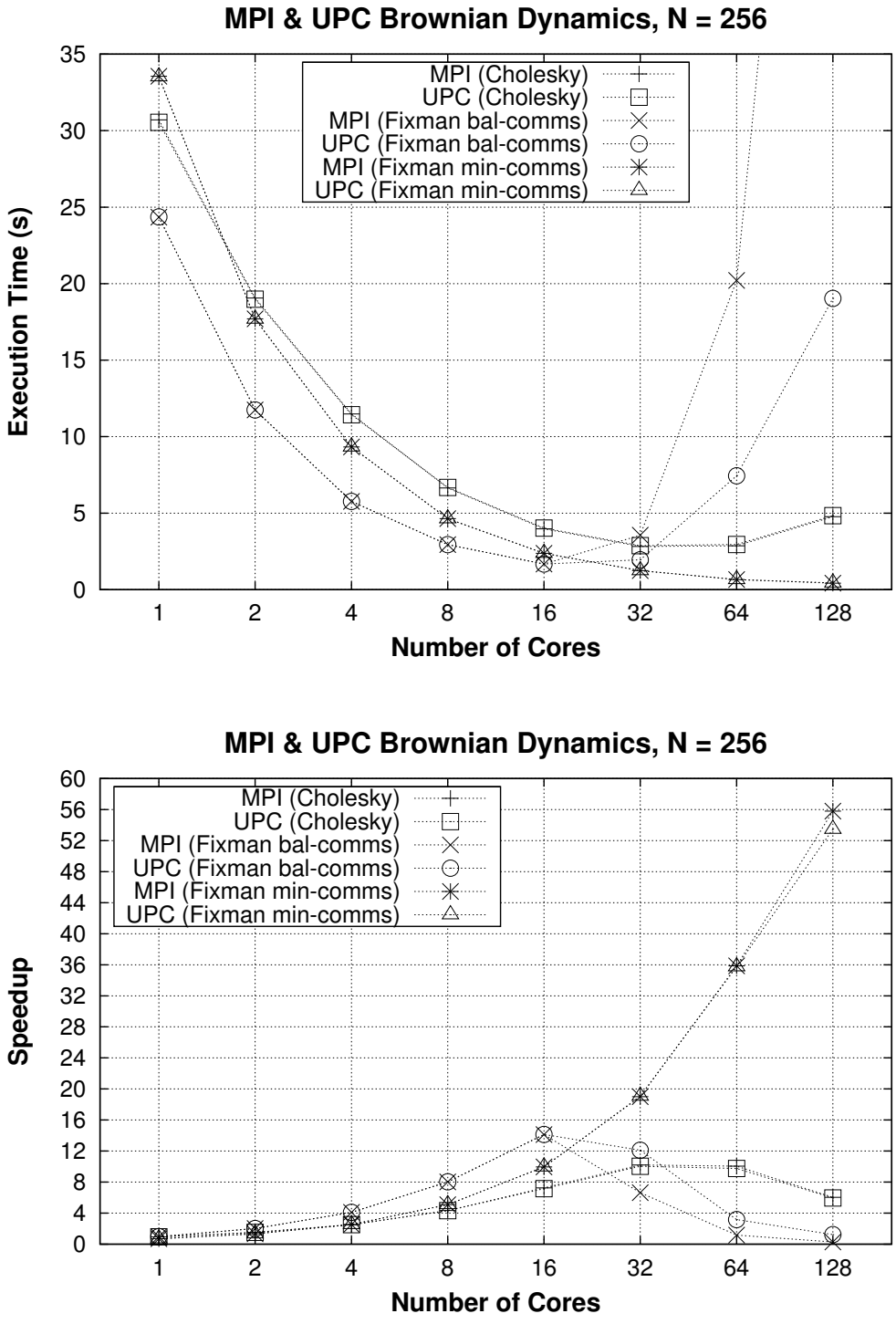


Figure 5.8: Distributed memory performance results with 256 particles on JRP

the number of particles than in the codes based on Fixman (see Table 5.1), confirming that Fixman is the best choice for the generation of random displacements, hence discarding the use of Cholesky for the larger problem sizes of Figures 5.9 and 5.10.

Comparing MPI and UPC *bal-comms* codes when using 32 or more cores, it can be observed that the overhead of the all-to-all communications, which is due both to memory requirements for internal buffering and synchronization costs, becomes an important performance bottleneck, in particular for MPI. Regarding UPC, the implementation of the extended *priv* in-place exchange function (see Section 3.2.1) manages memory and synchronizations more efficiently.

Figures 5.9 and 5.10 present the performance results with 50 time steps using 1024 and 4096 particles, respectively. In general, the results of the simulations are very similar to those of Figure 5.8, so an analogous interpretation can also be applied here: on the one hand, the *bal-comms* version obtains an almost linear speedup up to 32 cores for 1024 particles, and up to 64 cores for 4096 particles. Additionally, *bal-comms* obtains the best results up to the number of cores for which the computation time is still higher than the communication time (i.e., up to 32 cores for 1024 particles, and up to 64 cores with MPI and 128 cores with UPC for 4096 particles), and again UPC all-to-all communications represent a better choice than MPI in the simulation. On the other hand, *min-comms* shows the highest scalability, both for MPI and UPC, achieving in general a speedup of about half of the number of cores being used (i.e., a parallel efficiency of around 50%). Taking into account that this implementation requires almost double the number of computations of the original sequential code (hence its speedup with one core is around 0.6), this represents a significant scalability. Furthermore, the *min-comms* results in Figure 5.10 show a slight difference between MPI and UPC for 1024 and 2048 cores, mainly caused by the shared memory cache coherence mechanisms of the UPC runtime, whose implementation presents a significant overhead when handling thousands of cores.

Another factor to be considered is the number of particles handled per core, which limits the exploitation of each parallel solution. For example, according to the UPC performance results of Figures 5.8-5.10 and assuming that all time steps in a simulation have the same workload, the *bal-comms* version generally obtains the best results when 16 or more particles per core are simulated in a system with

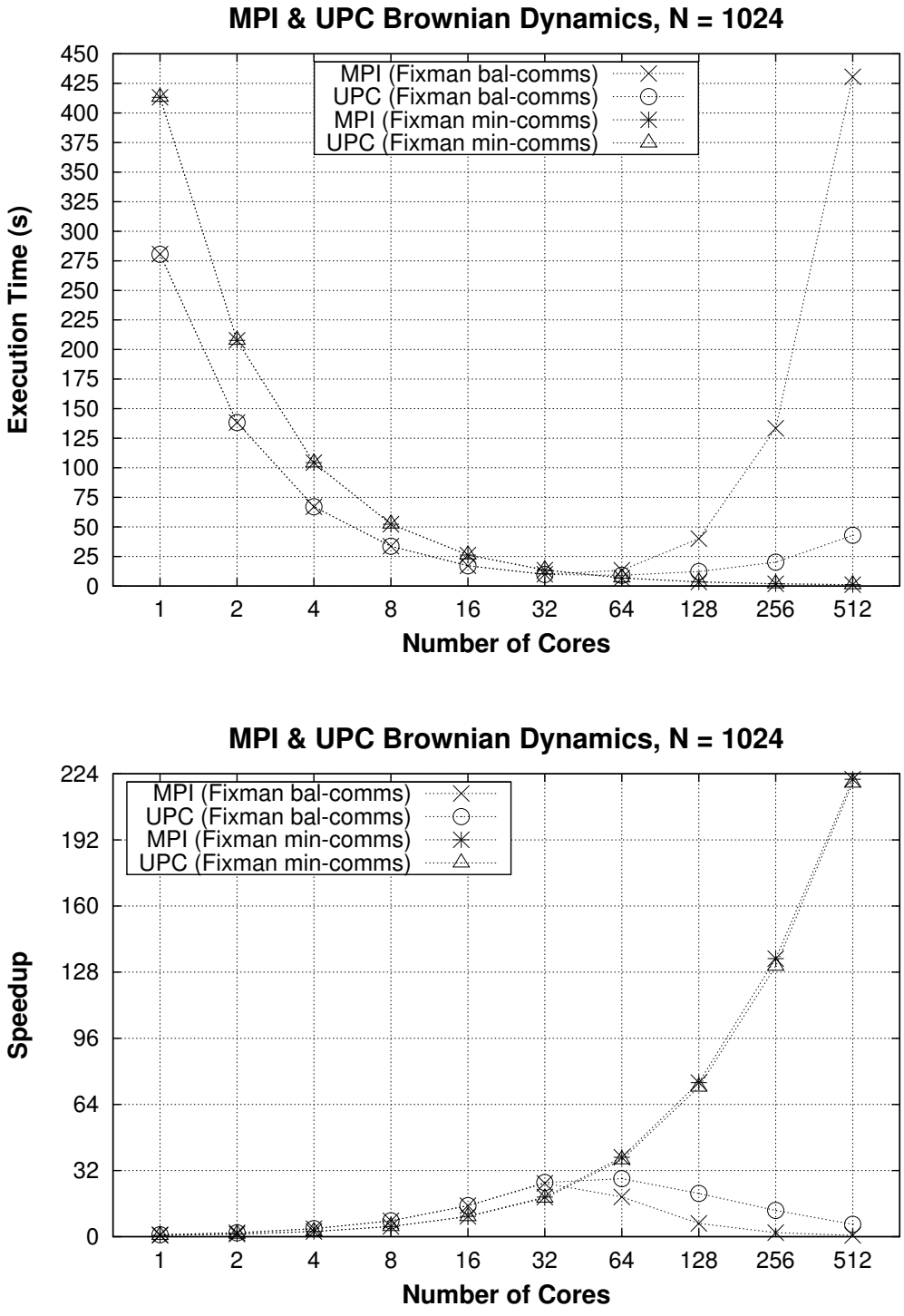


Figure 5.9: Distributed memory performance results with 1024 particles on JRP

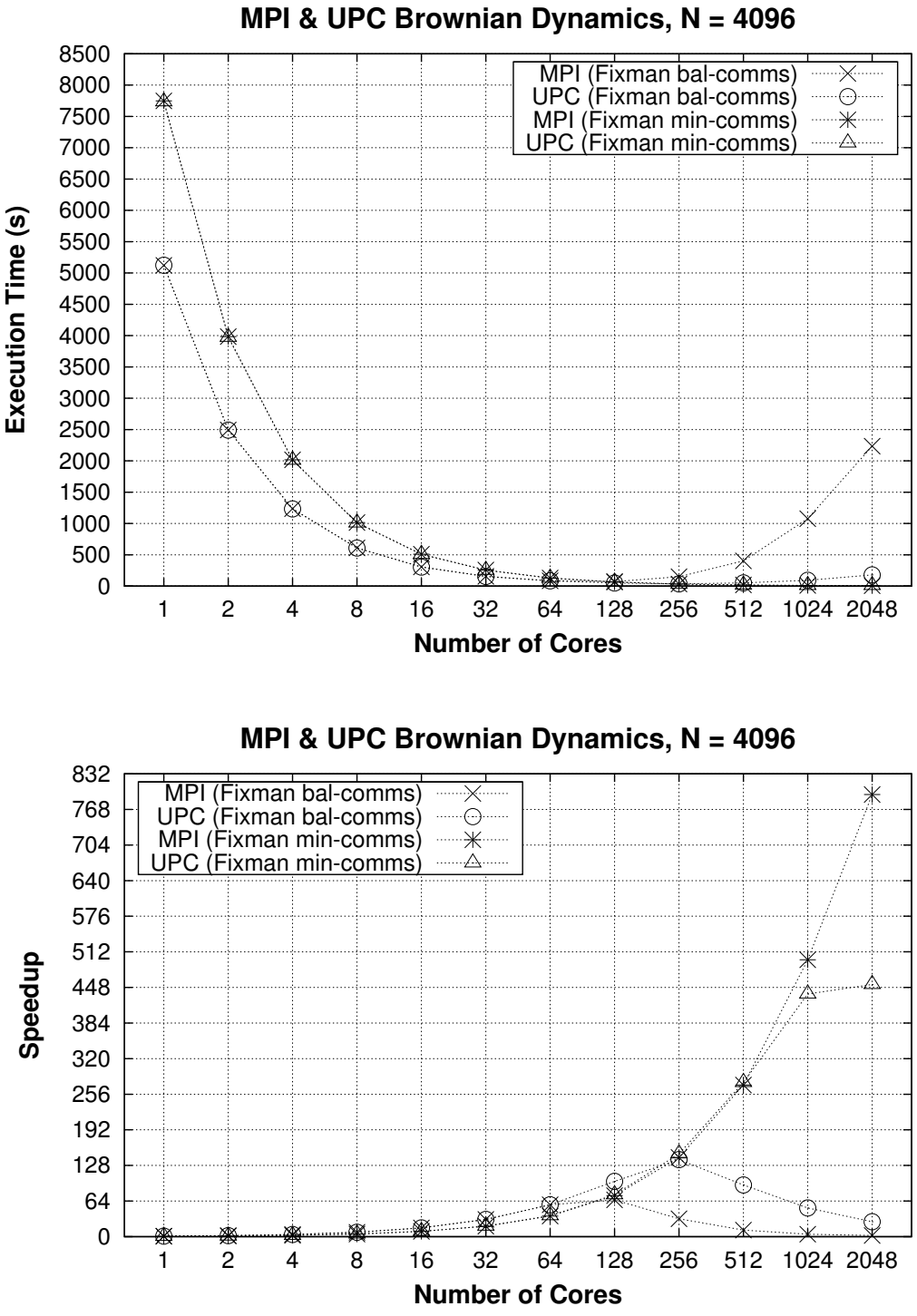


Figure 5.10: Distributed memory performance results with 4096 particles on JRP

256 particles (that is, when 16 or less cores are used); for simulations of 1024 and 4096 particles, *bal-comms* obtains the best results with 32 or more particles per core. This situation is illustrated in Figure 5.11 in the range of 16–256 cores, where the efficiency of the *bal-comms* algorithm for all problem sizes can be seen in relative terms compared to the performance of the *min-comms* counterpart: the higher the percentage is, the more efficient the algorithm is for the corresponding problem size. These results indicate that the *bal-comms* approach is overall a good choice when using more than 16-32 particles per core, i.e. scenarios where the memory is limited (because of the optimized storage of matrix `D` commented in Section 5.4.4) and the ratio computation/communication time is high, regardless of the actual number of cores or particles used in the simulation.

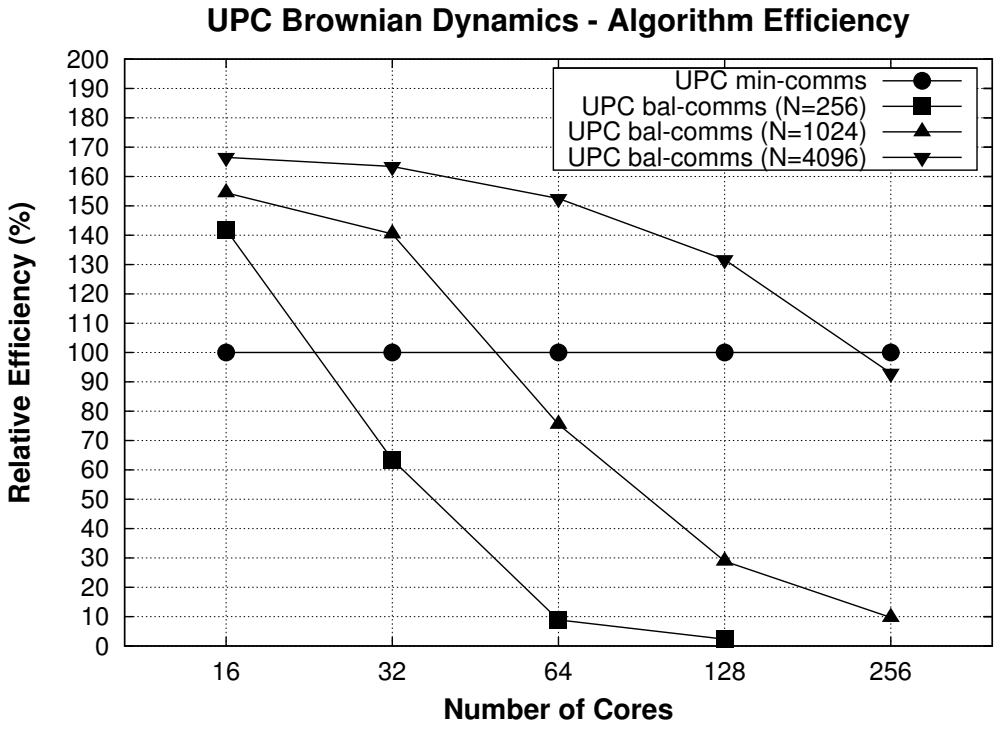


Figure 5.11: Efficiency comparison between Fixman *bal-comms* and *min-comms* algorithms, expressed as the percentage of the efficiency of the *min-comms* code

5.6. Conclusions to Chapter 5

Brownian dynamics simulations are a very relevant tool to analyze particle interactions under certain conditions, and also represent a computationally-intensive task that requires an efficient and scalable parallelization, in order to study large particle systems on high performance supercomputers under representative constraints. Thus, this chapter has presented the design and implementation of a parallel Brownian dynamics simulation using different approaches for shared and distributed memory environments using OpenMP, MPI and UPC. The main contributions of this parallelization are: (1) the analysis of data dependencies in the simulation codes and the domain decompositions for different environments, (2) the assessment of the alternatives in the work distributions to maximize performance and manage memory requirements efficiently, and (3) the performance evaluation of different versions of the parallel code on representative supercomputers with a large number of cores.

The experimental results have shown that codes using Fixman's algorithm outperform codes with Cholesky decomposition for all scenarios, and that there is no single optimal approach on distributed memory: the balanced communications version (*bal-comms*) presents the best performance when computation time is higher than communication time, whereas a more scalable approach (*min-comms*) can take advantage of a higher number of cores. Thus, the implemented approaches, both with MPI and UPC, are able to scale performance up to thousands of cores (using *min-comms*) while providing an alternative implementation with less memory requirements for a reduced number of cores (using *bal-comms*). Regarding the programming models considered, significant differences have been found. The higher maturity of MPI routines has provided high performance (showing the best results when using more than 1024 cores) at the cost of a higher programming effort. OpenMP has provided the lowest time to solution, providing a good approach for intranode communications on shared memory. Finally, UPC allowed the development of a quick parallel prototype thanks to its PGAS model, but required additional efforts to obtain higher performance. Nevertheless, UPC has been able to merge the shared and distributed memory approaches on a single code, outperforming significantly the scalability obtained with the OpenMP approach and being able to rival MPI in performance (and even beating its MPI *bal-comms* counterpart code when

communications are the main bottleneck). These UPC results have also been related to the use of extended collective functions, which take advantage of the use of one-sided operations on a hybrid shared/distributed memory architecture and help to minimize the communication overhead.

Conclusions

This PhD Thesis, *"Extended Collectives Library for Unified Parallel C"*, has analyzed the current developments and features provided by the UPC language, an extension of ANSI C designed for parallel programming, focusing on the development of libraries to enhance the productivity of UPC programming. The PGAS memory model has been proved to provide programmability for HPC applications, thus the focus of the Thesis has been put on assessing the usefulness of PGAS facilities in UPC, and providing additional libraries for an efficient and productive code development.

In order to assess the tradeoffs between benefits and drawbacks of the actions that have to be carried out in the improvement of UPC productivity, the first stage in the development of this Thesis has been the analysis of the UPC language in terms of performance and programmability. The main sources of information in these areas have been the microbenchmarking of UPC functions and the evaluation of programmability using classroom studies with two heterogeneous groups of programmers. The analysis of the codes developed by the participants in the programmability sessions and their feedback about the language, as well as a general evaluation of UPC language facilities, have reported different facts about UPC that have helped to define some areas of improvement:

- The achievement of the highest performance tends to conflict with the use of programmable constructs: some language facilities for parallel programming, such as direct assignments to shared variables, do not obtain an optimal performance, and they are substituted by more efficient workarounds (e.g., variable privatizations), which introduce higher complexity in the code. The most convenient solution to this would be an efficient compiler-based implementation

of these data transfers, but the use of small remote data transfers is generally difficult to optimize.

- Some of the functions in the standard language libraries, namely the collective primitives, present low flexibility, as there are multiple restrictions in their applicability to parallel codes. For example, all collective functions in the UPC specification can only be used with shared variables, and the amount of data involved has to be the same for all threads. An additional side effect is that privatized arrays cannot be used on collective communications, thus performance implications are also present here.

- There is a general lack of developments focusing on programmability: the improvement of language implementations at low level has been the main target for UPC developments in the last years, and programmability has been assumed as an implicit factor in the language. Even though this fact applies when UPC is compared to other parallel programming approaches (e.g., MPI), UPC developments have not explored the possibilities of providing additional language libraries or high-level frameworks to simplify code writing.

As a result of these facts, this Thesis has focused on giving a solution to these problems in terms of programmability, while not disregarding performance. Consequently, the second stage of this Thesis has been the improvement of the existing UPC programmability facilities, specifically the extension of the UPC collectives library. This extended library has overcome the most relevant limitations by providing the following additional features:

- The use of the same array as source and destination of communications (in-place collectives).

- The support of arrays of variable size as source and destination of the communications in each thread (vector-variant collectives).

- The selection of a subset of all threads in a UPC program in order to execute a specific task (team-based collectives).

- The use of private arrays as source and/or destination of communications (get-put-priv collectives).

In total, the library includes about 200 extended collective primitives that implement the previous features, including several variations and combinations of functionalities, aiming to give a complete support that maximizes the flexibility of the standard UPC collectives library. Internally, the algorithms for these functions are implemented efficiently. The main features of the functions in this library are:

- Implementation adapted to the execution on multicore architectures: the availability of different algorithms for collective communications allows the selection of the optimal one for each architecture configuration. Shared memory communications are exploited efficiently using data transfer schemes based on flat trees, whereas internode communications take advantage of binomial trees for data transfers. Different mixed approaches are also provided in order to adapt to the target environment, using thread pinning whenever possible and memory usage optimizations.

- Internal privatization of arguments to extended collectives: the definition of source and destination arrays, as well as other auxiliary arguments, in the interface of many collective functions uses shared variables; for efficiency purposes, the implemented functions privatize the access to these data.

- Development of a library-based support for teams: the use of team-based collectives has to be built on top of an implementation of teams, but no standard implementation has been built yet. In order to solve this issue, an interface for the management of teams has been defined, including functions to create and destroy teams, as well as for thread insertion and removal. A reference implementation for these functions is given in order to provide the required functionality for extended collectives.

The use of the extended collectives library has been initially tested using four representative kernel codes, which perform common operations such as matrix multiplication (dense and sparse), integer sort or a 3D Fast Fourier Transform (FFT). The flexibility of the developed library has allowed the introduction of collective communications in these codes replacing the original raw data memory copies and the explicit privatization of variables, thus giving out simpler codes. Additionally,

the multicore-aware algorithms and optimizations included in the library have improved the performance of UPC codes, which in some cases have shown to be clearly more efficient than the MPI counterparts, such as in the case of the 3D FFT.

The third stage in the development of the Thesis has been the implementation of a new functionality focused on programmability in UPC, which has led to the development of the UPC MapReduce (UPC-MR) framework. UPC-MR represents the adaptation of a popular framework for coarse-grain parallelism to the PGAS paradigm, and it has been conceived for programmability by assuming some basic principles:

- The user does not need to deal with parallel processing: two template management functions for the "Map" and the "Reduce" stages, respectively, are defined as user interface, performing all communications between threads transparently to the user.
- The implementation is generic: all the parameters of the management functions are processed analogously, regardless of their data type. The only necessary requirement is that the functions defined by the user for the "Map" and the "Reduce" stages have to match the given input elements.
- The processing is optimized according to the communication requirements: if some fine tuning is required, UPC-MR provides enough flexibility to configure its operation. The use of different configuration flags and the extended collectives library improves the data management and the communications between threads, respectively.

Finally, the last stage in the development of this Thesis has been the analysis and evaluation of large applications that take advantage of the previous implementations (UPC extended collectives and the UPC-MR framework) in high-end computer systems, in order to test their scalability with a large number of threads (up to 2048). This evaluation has included: (1) different MapReduce codes, and (2) the parallelization of a Brownian dynamics simulation.

The applications used to test the UPC-MR framework present different computational weights for the "Map" and "Reduce" stages, as well as a different number of

input elements, in order to analyze the impact of these features on the performance of the test codes and compare them with the MapReduce counterparts on shared and distributed memory using C (Phoenix) and MPI (MapReduce-MPI), respectively. The main conclusions extracted from the evaluation are:

- The amount of processing associated to each input element at the "Map" stage is a key factor for the overall performance: if it is too small, the element-by-element processing in the MPI implementation becomes significantly more inefficient than the UPC approach on distributed memory, especially when the number of input elements is very large. The shared memory code with Phoenix obtains better results, but also lower than UPC because of the repeated use of C++ library calls.

- The use of extended collectives can help to optimize performance. This applies especially for a large number of threads (i.e., when the number of communications at the "Reduce" stage becomes large), with the possibility of providing scalability up to thousands of cores depending on the relative time consumption of this stage with respect to the corresponding mapping function.

Regarding the simulation of Brownian dynamics, three implementations have been developed for comparison purposes, using OpenMP, MPI and UPC. As a general fact, the OpenMP implementation offers the highest programmability through the use of directives, although its performance is limited to the computational power of a single node, not taking advantage of distributed memory systems, similarly to the direct shared memory parallelization with UPC. Nevertheless, a fine tuning of the data distribution allows MPI and UPC to obtain quite good performance and scalability. The main outcomes of this work are:

- The analysis of the sequential code and its data dependencies, which helped to detect the core parts of the simulation algorithm and guided the domain decomposition process on each execution environment.

- The use of the developed extended collective functions, which have improved UPC performance scaling up to thousands of cores.

- A single UPC implementation has been able to achieve high performance on both shared and distributed memory environments, obtaining generally similar or even higher performance than OpenMP and MPI, respectively.

The contributions of this Thesis have been published in several peer-reviewed journals and conferences. The initial evaluations of UPC performance [44, 46, 77] and programmability [83] have led to the design and implementation of the extended collectives library [82], whose functions have been successfully applied to an efficient UPC implementation of the MapReduce framework [81] and to the parallelization of the Brownian dynamics simulation [78, 79, 80].

Among the main research lines to extend this work, the integration of the proposed functionalities in the currently available compilers deserves to be mentioned as a first step for its acceptance and spread in the UPC community. The forthcoming UPC standard specifications will provide a new framework for the development of new facilities for the UPC language, and at this point the implemented extended collectives can be used as a reference implementation that illustrates the benefits of a wider range of collective functions. As a result, and also having proved the high performance of the codes developed and evaluated in this Thesis, it will be possible to discuss the integration of the library (or a subset of it) as a required or optional library in future UPC specifications.

Appendix A

API of the UPC Extended Collectives Library

This appendix shows the complete description of the extended collectives library presented in Chapter 3, showing the details of all the functions following a man-page-like approach, with a similar format to the UPC language specification documents [93]. As a result, each subsection of this chapter is not directly related with the other subsections, and could be read independently.

The descriptions are structured in four main groups according to their functionality: in-place, vector-variant, team-based and get-put-priv functions. The latter group provides a wide variety of combinations for the other groups of functions (e.g., priv in-place or put vector-variant functions), therefore these operations are referenced in the same subsection as the corresponding base function, alongside with other possible variants (see the description of the library in Section 3.2 for more details).

All extended collectives are described using references to the eight functions in the standard UPC library and to a new allreduce collective (as mentioned in Section 3.1). This collective presents the same arguments as the standard UPC reduce, but here all threads get the final result.

A.1. In-place Collectives

A.1.1. The upc_all_broadcast_in_place Collective

Synopsis

```
#include <upc.h>
void upc_all_broadcast_in_place (
        shared void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_broadcast_in_place_priv (
        void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_broadcast_rooted_in_place (
        shared void *buffer, size_t nbytes, int root, upc_flag_t sync_mode
);
void upc_all_broadcast_rooted_in_place_priv (
        void *buffer, size_t nbytes, int root, upc_flag_t sync_mode
);
```

Description

- This is a UPC broadcast collective that uses the same array as source and destination of communications.
- The `upc_all_broadcast_in_place` function copies a block of shared memory with affinity to a thread to another block of shared memory on each thread using the same array as source and destination. The number of bytes in each block is `nbytes`, and it must be strictly greater than zero.
- The `upc_all_broadcast_rooted_in_place` is analogous to the previous one, but includes a parameter called `root` to indicate the thread that is used as source for communications. Additionally, pointer `buffer` must have affinity to thread 0.
- Both functions treat the `buffer` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes] char [nbytes*THREADS]
  ```

Priv Variants

- A variant called *priv* is defined by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. Thread 0 is considered as the root thread, therefore the private `buffer` associated to it is considered as source for communications with this collective.
- A *priv* variant is also defined for the `_rooted_in_place` function. Analogously to the base collective, the private array associated to thread `root` is considered as source for communications with this collective.
- Apart from the commented parameters, all of these variants present the same additional arguments as the base collective.

Examples

In the next piece of code, the whole shared memory block associated to thread 0 in array `A` is copied to the rest of blocks associated to all threads in the same array. After that, the block associated to thread 1 is copied to the rest of threads.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_broadcast_in_place(A, NELEMS*sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_broadcast_rooted_in_place(A, NELEMS*sizeof(int), 1,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.1.2. The `upc_all_scatter_in_place` Collective

Synopsis

```
#include <upc.h>
```

```
void upc_all_scatter_in_place (
        shared void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_scatter_in_place_priv (
        void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_scatter_rooted_in_place (
        shared void *buffer, size_t nbytes, int root, upc_flag_t sync_mode
);
void upc_all_scatter_rooted_in_place_priv (
        void *buffer, size_t nbytes, int root, upc_flag_t sync_mode
);
```

Description

- This is a UPC scatter collective that uses the same array as source and destination of communications.
- The `upc_all_scatter_in_place` function copies the `i`th portion of a block of shared memory with affinity to a thread to another block of shared memory with affinity to the `i`th thread in the same array. The number of bytes in each block is `nbytes`, and it must be strictly greater than zero.
- The `upc_all_scatter_rooted_in_place` is analogous to the previous one, but includes a parameter called `root` to indicate the thread that is used as source for communications. Additionally, pointer `buffer` must have affinity to thread 0.
- Both functions treat the `buffer` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]
  ```

Priv Variants

- A variant called *priv* is defined by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. Thread 0 is considered as the root thread, therefore the private `buffer` associated to it is considered as source for communications with this collective.

- A *priv* variant is also defined for the `_rooted_in_place` function. Analogously to the base collective, the private array associated to thread `root` is considered as source for communications with this collective.
- Apart from the commented parameters, all of these variants present the same additional arguments as the base collective.

Examples

In the next piece of code, the shared memory block associated to thread 0 in array `A` is scattered to the rest of threads. After that, the memory block associated to thread 1 is scattered.

```
#define NELEMS 10
shared [NELEMS*THREADS] int A[NELEMS*THREADS*THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS*THREADS; i<(MYTHREAD+1)*NELEMS*THREADS; i++) {
    A[i]=i;
}

upc_all_scatter_in_place(A, NELEMS*sizeof(int), UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_scatter_rooted_in_place(A, NELEMS*sizeof(int), 1,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.1.3. The `upc_all_gather_in_place` Collective

Synopsis

```
#include <upc.h>
void upc_all_gather_in_place (
         shared void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_in_place_priv (
         void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_rooted_in_place (
         shared void *buffer, size_t nbytes, int root, upc_flag_t sync_mode
);
```

```
void upc_all_gather_rooted_in_place_priv (
        void *buffer, size_t nbytes, int root, upc_flag_t sync_mode
);
```

Description

- This is a UPC gather collective that uses the same array as source and destination of communications.
- The `upc_all_gather_in_place` function copies a block of shared memory with affinity to the `i`th thread to the `i`th portion of a block of shared memory with affinity to a thread in the same array. The number of bytes in each block is `nbytes`, and it must be strictly greater than zero.
- The `upc_all_gather_rooted_in_place` is analogous to the previous one, but includes a parameter called `root` to indicate the thread that is used as source for communications. Additionally, pointer `buffer` must have affinity to thread 0.
- Both functions treat the `buffer` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]
  ```

Priv Variants

- A variant called *priv* is defined by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. Thread 0 is considered as the root thread, therefore the private `buffer` associated to it is considered as destination for communications with this collective.
- A *priv* variant is also defined for the `_rooted_in_place` function. Analogously to the base collective, the private array associated to thread `root` is considered as destination for communications with this collective.
- Apart from the commented parameters, all of these variants present the same additional arguments as the base collective.

Examples

In the next piece of code, the shared memory blocks in array `A` are gathered in the memory block with affinity to thread 0 in the same array. After that, the same blocks are gathered in the shared memory of thread 1.

```
#define NELEMS 10
shared [NELEMS*THREADS] int A[NELEMS*THREADS*THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS*THREADS; i<(MYTHREAD+1)*NELEMS*THREADS; i++) {
    A[i]=i;
}

upc_all_gather_in_place(A, NELEMS*sizeof(int), UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_gather_rooted_in_place(A, NELEMS*sizeof(int), 1,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.1.4. The `upc_all_gather_all_in_place` Collective

Synopsis

```
#include <upc.h>
void upc_all_gather_all_in_place (
         shared void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_all_in_place_priv (
         void *buffer, size_t nbytes, upc_flag_t sync_mode
);
```

Description

- This is a UPC allgather collective that uses the same array as source and destination of communications.
- The `upc_all_gather_all_in_place` function copies a block of shared memory with affinity to the `i`th thread to the `i`th portion of a shared memory area on each thread in the same array. The number of bytes in each block is `nbytes`, and it must be strictly greater than zero.

- The `upc_all_gather_all_in_place` function treats the `buffer` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]
  ```

- The target of the `buffer` pointer must have affinity to thread 0. Additionally, this pointer is treated as if it had phase 0.

Priv Variants

- A variant called *priv* is defined by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis.
- Apart from the source/destination pointer, this variant presents the same additional arguments as the base collective.

Example

In the next piece of code, the shared memory blocks in array `A` are gathered in the memory blocks with affinity to every thread using the same array.

```
#define NELEMS 10
shared [NELEMS*THREADS] int A[NELEMS*THREADS*THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS*THREADS; i<(MYTHREAD+1)*NELEMS*THREADS; i++) {
    A[i]=i;
}

upc_all_gather_all_in_place(A, NELEMS*sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.1.5. The `upc_all_exchange_in_place` Collective

Synopsis

```
#include <upc.h>
```

```
void upc_all_exchange_in_place (
        shared void *buffer, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_exchange_in_place_priv (
        void *buffer, size_t nbytes, upc_flag_t sync_mode
);
```

Description

- This is a UPC exchange collective that uses the same array as source and destination of communications.
- The `upc_all_exchange_in_place` function copies the `i`th block of shared memory with affinity to the `j`th thread to the `j`th portion of a shared memory area associated to the `i`th thread in the same array. The number of bytes in each block is `nbytes`, and it must be strictly greater than zero.
- The `upc_all_exchange_in_place` function treats the `buffer` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]
  ```

- The target of the `buffer` pointer must have affinity to thread 0. Additionally, this pointer is treated as if it had phase 0.

Priv Variants

- A variant called *priv* is defined by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis.
- Apart from the source/destination pointer, this variant presents the same additional arguments as the base collective.

Example

In the next piece of code, the shared memory blocks in array `A` are exchanged between threads using the same array.

```
#define NELEMS 10
shared [NELEMS*THREADS] int A[NELEMS*THREADS*THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS*THREADS; i<(MYTHREAD+1)*NELEMS*THREADS; i++) {
    A[i]=i;
}

upc_all_exchange_in_place(A, NELEMS*sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.1.6. The upc_all_permute_in_place Collective

Synopsis

```
#include <upc.h>
void upc_all_permute_in_place (
         shared void *buffer, shared const int *perm, size_t nbytes,
         upc_flag_t sync_mode
);
void upc_all_permute_in_place_priv (
         void *buffer, shared const int *perm, size_t nbytes,
         upc_flag_t sync_mode
);
```

Description

- This is a UPC permute collective that uses the same array as source and destination of communications.
- The `upc_all_permute_in_place` function copies a block of shared memory with affinity to the `i`th thread to a portion of a shared memory area associated to thread `perm[i]` in the same array. The number of bytes in each block is `nbytes`, and it must be strictly greater than zero.
- The values stored in `perm[0...THREADS-1]` must define a permutation of the integer values between `0` and `THREADS-1`.
- The `upc_all_permute_in_place` function treats the `buffer` pointer as if it pointed to a shared memory area with type:

```
shared [nbytes] char [nbytes*THREADS]
```

- The targets of the `buffer` and `perm` pointers must have affinity to thread 0. Additionally, the `buffer` pointer is treated as if it had phase 0.

Priv Variants

- A variant called *priv* is defined by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis.

- Apart from the source/destination pointer, this variant presents the same additional arguments as the base collective.

Example

In the next piece of code, the shared memory blocks in array `A` are permuted in the same array according to the values in `perm`. This code is specifically designed for 4 threads.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared int perm[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

// Define the desired permutation
perm[0]=3;
perm[1]=2;
perm[2]=0;
perm[3]=1;

upc_all_permute_in_place(A, perm, NELEMS*sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.1.7. Computational In-place Collectives

Synopsis

```
#include <upc.h>
#include <upc_collective.h>
/**
 * REDUCE
 */
void upc_all_reduce<<T>>_in_place (
         shared void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_in_place_priv (
         void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_rooted_in_place (
         shared void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
         int root, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_rooted_in_place_priv (
         void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
         int root, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
/**
 * ALLREDUCE
 */
void upc_all_reduce<<T>>_all_in_place (
         shared void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_in_place_priv (
         void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
/**
 * PREFIX REDUCE
 */
void upc_all_prefix_reduce<<T>>_in_place (
         shared void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
```

```
        <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_prefix_reduce<<T>>_in_place_priv (
        void *buffer, upc_op_t op, size_t nelems, size_t blk_size,
        <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
```

Description

- These functions define a set of UPC computational operations that use the same array as source and destination of communications.
- The prototypes above represent 88 computational extended collectives, where *T* and *TYPE* have the following correspondences:

T	*TYPE*	*T*	*TYPE*
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

For example, if *T* is C, *TYPE* must be signed char

- The reduce and allreduce functions present the same arguments as the standard UPC reduction operation, but using a single array as source and destination (buffer). The upc_all_reduce*T*_rooted_in_place function also presents the parameter that determines the root thread for communications (root).
- The prefix reduce function presents the same arguments as the standard UPC prefix reduction operation, but using a single array as source and destination (buffer).
- On completion of the upc_all_reduce*T*_in_place functions, the value of the *TYPE* shared variable referenced by buffer is set to buffer[0] $\oplus$ buffer[1] $\oplus \cdots \oplus$ buffer[nelems-1], where "$\oplus$" is the operator specified by the op parameter.
- On completion of the upc_all_reduce*T*_all_in_place functions, the value of the *TYPE* shared variable referenced by buffer[i*blk_size] for each thread i is set to buffer[0] $\oplus$ buffer[1] $\oplus \cdots \oplus$ buffer[nelems-1], where "$\oplus$" is the operator specified by the op parameter.

- On completion of the `upc_all_prefix_reduce`*T*`_in_place` functions, the value of the *TYPE* shared variable referenced by `buffer[i]` is set to `buffer[0]` $\oplus$ `buffer[1]` $\oplus$ $\cdots$ $\oplus$ `buffer[i]` for `0`$\leq$`i`$\leq$`nelems-1`, where "$\oplus$" is the operator specified by the `op` parameter.

- The argument `op` can have the following values:

  - `UPC_ADD`: addition.
  - `UPC_MULT`: multiplication.
  - `UPC_AND`: bitwise `AND` for integer and character variables. Results are undefined for floating point numbers.
  - `UPC_OR`: bitwise `OR` for integer and character variables. Results are undefined for floating point numbers.
  - `UPC_XOR`: bitwise `XOR` for integer and character variables. Results are undefined for floating point numbers.
  - `UPC_LOGAND`: logical `AND` for all variable types.
  - `UPC_LOGOR`: logical `OR` for all variable types.
  - `UPC_MIN`: for all data types, find the minimum value.
  - `UPC_MAX`: for all data types, find the maximum value.
  - `UPC_FUNC`: use the specified commutative function `func` to operate on the data in the `buffer` array.
  - `UPC_NONCOMM_FUNC`: use the specified non-commutative function `func` to operate on the data in the `buffer` array.

- The operations represented by `op` are assumed to be associative and commutative (except those provided using `UPC_NONCOMM_FUNC`). An operation whose result is dependent on the operator evaluation or on the order of the operands will have undefined results.

- If the value of `blk_size` passed to these functions is greater than zero, they treat the `buffer` pointer as if it pointed to a shared memory area of `nelems` elements of type *TYPE* and blocking factor `blk_size`, and therefore with type:

```
shared [nbytes] <<TYPE>> [nelems]
```

- If the value of `blk_size` passed to these functions is zero, they treat the `buffer` pointer as if it pointed to a shared memory area of `nelems` elements of type `TYPE` with an indefinite layout qualifier, and therefore with type:

  ```
  shared [] <<TYPE>> [nelems]
  ```

- It is important to note that these functions overwrite one of the values used to perform the computation in order to store the final result of each function call. Therefore, if two consecutive calls to these functions are performed, their results are not likely to be the same.

Priv Variants

- The *priv* variant is defined for these collectives by using a private pointer as target for communication. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. Thread 0 is considered as the root thread, therefore the private `buffer` associated to it is considered as source for communications with this collective.
- A *priv* variant is also defined for the `_rooted_in_place` reduction. Analogously to the base collective, the private array associated to thread `root` is considered as source for communications with this collective.
- Apart from the commented parameters, all of these variants present the same additional arguments as the base collective.

Examples

In the next piece of code, four examples of the presented collectives are shown, considering the use of two or more threads. First, the sum of all the integer elements in array `A` is computed and stored in the element with affinity to thread 0 in the same array. Next, the sum of elements `[1000...1999]` in `A` is stored in the shared memory associated to thread 1. After that, the sum of the first 1000 elements in `A` is computed and sent to all threads. Finally, the accumulative sum of all the elements in `A` is stored in the same array.

```
#define NELEMS 1000
shared [NELEMS] int A[NELEMS*THREADS];
```

```
// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_reduceI_in_place(A, UPC_ADD, NELEMS*THREADS, THREADS, NULL,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_reduceI_rooted_in_place(A, UPC_ADD, NELEMS, THREADS, 1, NULL,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_reduceI_all_in_place(A, UPC_ADD, NELEMS, THREADS, NULL,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_prefix_reduceI_in_place(A, UPC_ADD, NELEMS*THREADS, THREADS, NULL,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2. Vector-variant Collectives

A.2.1. The upc_all_broadcast_v Collective

Synopsis

```
#include <upc.h>
void upc_all_broadcast_v (
         shared void *dst, shared const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_local (
         shared void *dst, shared const void *src, shared int *ddisp_local,
         size_t nelems, size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_raw (
         shared void *dst, shared const void *src, shared size_t *ddisp_raw,
         size_t nbytes, size_t dst_blk_raw, upc_flag_t sync_mode
);
void upc_all_broadcast_v_rooted (
         shared void *dst, shared const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, int root,
         upc_flag_t sync_mode
);
```

```
/**
 * GET VERSIONS
 */
void upc_all_broadcast_v_get (
         void *dst, shared const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_local_get (
         void *dst, shared const void *src, shared int *ddisp_local,
         size_t nelems, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_raw_get (
         void *dst, shared const void *src, shared size_t *ddisp_raw,
         size_t nbytes, size_t dst_blk_raw, upc_flag_t sync_mode
);
void upc_all_broadcast_v_rooted_get (
         void *dst, shared const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, int root,
         upc_flag_t sync_mode
);
/**
 * PUT VERSIONS
 */
void upc_all_broadcast_v_put (
         shared void *dst, const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_local_put (
         shared void *dst, const void *src, shared int *ddisp_local,
         size_t nelems, size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_raw_put (
         shared void *dst, const void *src, shared size_t *ddisp_raw,
         size_t nbytes, size_t dst_blk_raw, upc_flag_t sync_mode
);
void upc_all_broadcast_v_rooted_put (
         shared void *dst, const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, int root,
         upc_flag_t sync_mode
);
/**
```

```
 * PRIV VERSIONS
 */
void upc_all_broadcast_v_priv (
         void *dst, const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_local_priv (
         void *dst, const void *src, shared int *ddisp_local,
         size_t nelems, size_t typesize, upc_flag_t sync_mode
);
void upc_all_broadcast_v_raw_priv (
         void *dst, const void *src, shared size_t *ddisp_raw,
         size_t nbytes, size_t dst_blk_raw, upc_flag_t sync_mode
);
void upc_all_broadcast_v_rooted_priv (
         void *dst, const void *src, shared int *ddisp,
         size_t nelems, size_t dst_blk, size_t typesize, int root,
         upc_flag_t sync_mode
);
```

Description

- This is a UPC broadcast collective that uses a configurable amount of source and destination data per thread.
- The `upc_all_broadcast_v` function copies a piece of shared memory pointed by `src` to the different locations of the `dst` shared array on each thread indicated in `ddisp`. The `ddisp` array contains `THREADS` integers that correspond to the array indexes in `dst` (one per thread) to which each chunk of `nelems` elements of size `typesize` is copied. The parameter `nelems` must be strictly greater than zero. The number of elements in each block of the `dst` array is `dst_blk`.
- The `upc_all_broadcast_v_local` function is analogous to the previous one, but here the `ddisp_local` array should contain block phases to each thread instead of absolute array indexes in order to indicate a value for a thread. For example, if `ddisp_local[i]` is 1 in a call to `upc_all_broadcast_v_local`, the function will have the same behavior as with a call to `upc_all_broadcast_v` with `ddisp[i]` equal to `i*dst_blk+1`.
- The `upc_all_broadcast_v_raw` function indicates the amount of data for commu-

nications on each thread as a chunk of bytes of size `nbytes`. The displacements in the destination array are consequently described in terms of bytes in the parameter `ddisp_raw`, as well as in the block size parameter `dst_blk_raw`.

- The `upc_all_broadcast_v_rooted` function takes an extra argument (`root`) that indicates the thread used as root for communications.

- The chunk of shared memory that is broadcast must have affinity to only one thread. Analogously, the destination position for all threads must be less than or equal to the difference of `dst_blk` and `nelems` (`dst_blk_raw` and `nbytes` for `upc_all_broadcast_v_raw`). If any of these conditions is not fulfilled, the chunks received in array `dst` may be truncated.

- The `upc_all_broadcast_v` and `upc_all_broadcast_v_local` functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [] char [dst_blk*typesize]
  ```

  the `upc_all_broadcast_v_raw` function treats `src` as a pointer to a shared memory area with type:

  ```
  shared [] char [dst_blk_raw]
  ```

  and the `upc_all_broadcast_v_rooted` function treats `src` as a pointer to shared memory with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

- The `dst` pointer in all functions is considered as if it pointed to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  with `dst_blk*typesize = dst_blk_raw` for `upc_all_broadcast_v_raw`.

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective. The `dst_blk` and `dst_blk_raw` arguments, where present, represent a limit for the number of elements in each private destination array. It is important to note that in these functions the displacements in `ddisp/ddisp_raw` should be defined as if the destination array was a shared array with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  with `dst_blk*typesize = dst_blk_raw` for the `raw` function.

- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The shared destination pointers in each of these functions are treated the same way as in the corresponding base collective.

- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. The `dst_blk` and `dst_blk_raw` arguments, where present, represent a limit for the number of elements in each private destination array, and the block phases in `ddisp/ddisp_raw` are defined analogously to the *get* variant.

- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, a chunk of memory associated to thread 1 in array `A` is copied to the selected positions in array `B`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
```

```
shared int ddisp[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}
ddisp[MYTHREAD] = MYTHREAD*NELEMS + (MYTHREAD % (NELEMS/2));

upc_all_broadcast_v(B, &A[NELEMS+1], ddisp, 3, NELEMS, sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.2. The upc_all_scatter_v Collective

Synopsis

```
#include <upc.h>
void upc_all_scatter_v (
         shared void *dst, shared const void *src, shared int *ddisp,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_scatter_v_local (
         shared void *dst, shared const void *src, shared int *ddisp_local,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_scatter_v_raw (
         shared void *dst, shared const void *src, shared size_t *ddisp_raw,
         shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
         upc_flag_t sync_mode
);
void upc_all_scatter_v_rooted (
         shared void *dst, shared const void *src, shared int *ddisp,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         int root, upc_flag_t sync_mode
);
/**
 * GET VERSIONS
 */
void upc_all_scatter_v_get (
```

```
        void *dst, shared const void *src, shared int *ddisp,
        shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_scatter_v_local_get (
        void *dst, shared const void *src, shared int *ddisp_local,
        shared size_t *nelems, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_scatter_v_raw_get (
        void *dst, shared const void *src, shared size_t *ddisp_raw,
        shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
        upc_flag_t sync_mode
);
void upc_all_scatter_v_rooted_get (
        void *dst, shared const void *src, shared int *ddisp,
        shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        int root, upc_flag_t sync_mode
);
/**
 * PUT VERSIONS
 */
void upc_all_scatter_v_put (
        shared void *dst, const void *src, shared int *ddisp,
        shared size_t *nelems, size_t dst_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_scatter_v_local_put (
        shared void *dst, const void *src, shared int *ddisp_local,
        shared size_t *nelems, size_t dst_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_scatter_v_raw_put (
        shared void *dst, const void *src, shared size_t *ddisp_raw,
        shared size_t *nbytes, size_t dst_blk_raw,
        upc_flag_t sync_mode
);
void upc_all_scatter_v_rooted_put (
        shared void *dst, const void *src, shared int *ddisp,
        shared size_t *nelems, size_t dst_blk, size_t typesize,
        int root, upc_flag_t sync_mode
```

```
);
/**
 * PRIV VERSIONS
 */
void upc_all_scatter_v_priv (
         void *dst, const void *src, shared int *ddisp,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_scatter_v_local_priv (
         void *dst, const void *src, shared int *ddisp_local,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_scatter_v_raw_priv (
         void *dst, const void *src, shared size_t *ddisp_raw,
         shared size_t *nbytes, size_t dst_blk_raw,
         upc_flag_t sync_mode
);
void upc_all_scatter_v_rooted_priv (
         void *dst, const void *src, shared int *ddisp,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         int root, upc_flag_t sync_mode
);
```

Description

- This is a UPC scatter collective that uses a configurable amount of source and destination data per thread.
- The `upc_all_scatter_v` function copies `THREADS` chunks of shared memory from the address indicated in `src` to each of the `THREADS` array indexes in `dst` contained in `ddisp`. Each index in `ddisp` determines a destination position for a chunk of `nelems[i]` elements of size `typesize` for each thread `i`, which are obtained consecutively from `src`. The number of elements in a block in arrays `src` and `dst` is `src_blk` and `dst_blk`, respectively. All indexes in `ddisp` are absolute array indexes. All values in array `nelems` must be strictly greater than zero.
- The `upc_all_scatter_v_local` function is analogous to `upc_all_scatter_v`, but here the `ddisp_local` array should contain block phases to each thread instead of

absolute array indexes in order to indicate a value for a thread. For example, if `ddisp_local[i]` is 1 in a call to `upc_all_scatter_v_local`, the function will have the same behavior as with a call to `upc_all_scatter_v` with `ddisp[i]` equal to `i*dst_blk+1`.

- The `upc_all_scatter_v_raw` function indicates the amount of data for communications on each thread as a chunk of bytes following the sizes indicated by array `nbytes`. The displacements in the destination array are consequently described in terms of bytes in the parameter `ddisp_raw`, as well as in the block size parameters `dst_blk_raw` and `src_blk_raw`.

- The `upc_all_scatter_v_rooted` function takes an additional argument (`root`) that indicates the thread used as root for communications.

- The chunk of shared memory that is scattered must have affinity to only one thread. Analogously, the destination position for thread `i` must be less than or equal to the difference of `dst_blk` and `nelems[i]` (`dst_blk_raw` and `nbytes[i]` for `upc_all_scatter_v_raw`). If any of these conditions is not fulfilled, the chunks received in array `dst` may be truncated.

- The `src` pointer in `upc_all_scatter_v` and `upc_all_scatter_v_local` is interpreted as a pointer to a shared memory area with type:

  ```
  shared [] char [src_blk*typesize]
  ```

  the `upc_all_scatter_v_raw` function treats `src` as a pointer to a shared memory area with type:

  ```
  shared [] char [src_blk_raw]
  ```

  and the `upc_all_scatter_v_rooted` function treats `src` as a pointer to shared memory with type:

  ```
  shared [src_blk*typesize] char [src_blk*typesize*THREADS]
  ```

- The `dst` pointer in all functions is considered as if it pointed to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  with `dst_blk*typesize = dst_blk_raw` for `upc_all_scatter_v_raw`.

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective. The `dst_blk` and `dst_blk_raw` arguments, where present, represent a limit for the number of elements in each private destination array. It is important to note that in these functions the displacements in `ddisp`/`ddisp_raw` should be defined as if the destination array was a shared array with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  with `dst_blk*typesize = dst_blk_raw` for the `raw` function.

- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The shared destination pointers in each of these functions are treated the same way as in the corresponding base collective.

- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. The `dst_blk` and `dst_blk_raw` arguments, where present, represent a limit for the number of elements in each private destination array, and the block phases in `ddisp`/`ddisp_raw` are defined analogously to the *get* variant.

- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, the shared memory block associated to thread 1 in array `A` is scattered to the locations indicated in array `B`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
```

```
shared int ddisp[THREADS];
shared int nelems[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}
ddisp[MYTHREAD] = MYTHREAD*NELEMS + (MYTHREAD % (NELEMS/2));
nelems[MYTHREAD] = MYTHREAD % (NELEMS/2) + 1;

upc_all_scatter_v(B, &A[NELEMS+1], ddisp, nelems, NELEMS, NELEMS, sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.3. The upc_all_gather_v Collective

Synopsis

```
#include <upc.h>
void upc_all_gather_v (
         shared void *dst, shared const void *src, shared int *sdisp,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_gather_v_local (
         shared void *dst, shared const void *src, shared int *sdisp_local,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_gather_v_raw (
         shared void *dst, shared const void *src, shared size_t *sdisp_raw,
         shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
         upc_flag_t sync_mode
);
void upc_all_gather_v_rooted (
         shared void *dst, shared const void *src, shared int *sdisp,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         int root, upc_flag_t sync_mode
);
/**
 * GET VERSIONS
```

```
 */
void upc_all_gather_v_get (
         void *dst, shared const void *src, shared int *sdisp,
         shared size_t *nelems, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_gather_v_local_get (
         void *dst, shared const void *src, shared int *sdisp_local,
         shared size_t *nelems, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_gather_v_raw_get (
         void *dst, shared const void *src, shared size_t *sdisp_raw,
         shared size_t *nbytes, size_t src_blk_raw,
         upc_flag_t sync_mode
);
void upc_all_gather_v_rooted_get (
         void *dst, shared const void *src, shared int *sdisp,
         shared size_t *nelems, size_t src_blk, size_t typesize,
         int root, upc_flag_t sync_mode
);
/**
 * PUT VERSIONS
 */
void upc_all_gather_v_put (
         shared void *dst, const void *src, shared int *sdisp,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_gather_v_local_put (
         shared void *dst, const void *src, shared int *sdisp_local,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_gather_v_raw_put (
         shared void *dst, const void *src, shared size_t *sdisp_raw,
         shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
         upc_flag_t sync_mode
);
void upc_all_gather_v_rooted_put (
         shared void *dst, const void *src, shared int *sdisp,
```

```
        shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        int root, upc_flag_t sync_mode
);
/**
 * PRIV VERSIONS
 */
void upc_all_gather_v_priv (
        void *dst, const void *src, shared int *sdisp,
        shared size_t *nelems, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_gather_v_local_priv (
        void *dst, const void *src, shared int *sdisp_local,
        shared size_t *nelems, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_gather_v_raw_priv (
        void *dst, const void *src, shared size_t *sdisp_raw,
        shared size_t *nbytes, size_t src_blk_raw,
        upc_flag_t sync_mode
);
void upc_all_gather_v_rooted_priv (
        void *dst, const void *src, shared int *sdisp,
        shared size_t *nelems, size_t src_blk, size_t typesize,
        int root, upc_flag_t sync_mode
);
```

Description

- This is a UPC gather collective that uses a configurable amount of source and destination data per thread.

- The `upc_all_gather_v` function copies `THREADS` chunks of shared memory from each of the array indexes in `src` contained in `sdisp` to the address pointed by `dst`. Each index in `sdisp` determines a source position for a chunk of `nelems[i]` elements of size `typesize` for each thread `i`, which are merged consecutively from address `dst` on. The number of elements in a block in arrays `src` and `dst` is `src_blk` and `dst_blk`, respectively. All indexes in `sdisp` are absolute array indexes. All values in array `nelems` must be strictly greater than zero.

- The `upc_all_gather_v_local` function is analogous to `upc_all_gather_v`, but here the `sdisp_local` array should contain block phases to each thread instead of absolute array indexes in order to indicate a value for a thread. For example, if `sdisp_local[i]` is 1 in a call to `upc_all_gather_v_local`, the function will have the same behavior as with a call to `upc_all_gather_v` with `sdisp[i]` equal to `i*src_blk+1`.

- The `upc_all_gather_v_raw` function indicates the amount of data for communications on each thread as a chunk of bytes following the sizes indicated by array `nbytes`. The displacements in the source array are consequently described in terms of bytes in the parameter `sdisp_raw`, as well as in the block size parameters `src_blk_raw` and `dst_blk_raw`.

- The `upc_all_gather_v_rooted` function takes an additional argument (`root`) that indicates the thread used as root for communications.

- The chunk of shared memory that gathers all subarrays must have affinity to only one thread. Analogously, the source position for thread `i` must be less than or equal to the difference of `dst_blk` and `nelems[i]` (`dst_blk_raw` and `nbytes[i]` for `upc_all_gather_v_raw`). If any of these conditions is not fulfilled, the chunks received in array `dst` may be truncated.

- All these functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [src_blk*typesize] char [src_blk*typesize*THREADS]
  ```

  except the `upc_all_gather_v_raw` function, which treats `src` as a pointer to a shared memory area with type:

  ```
  shared [src_blk_raw] char [src_blk_raw*THREADS]
  ```

- The `dst` pointer in `upc_all_gather_v` and `upc_all_gather_v_local` is interpreted as a pointer to a shared memory area with type:

  ```
  shared [] char [dst_blk*typesize]
  ```

  and the same applies to `upc_all_gather_v_raw`, considering `dst_blk*typesize = dst_blk_raw`.

- The `dst` pointer in `upc_all_gather_v_rooted` is considered as if it pointed to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective. It is important to note that these functions do not require the argument `dst_blk`/`dst_blk_raw`, which is present in their corresponding base collectives.
- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The `src_blk` and `src_blk_raw` arguments, where present, represent a limit for the number of elements in each private source array. It is important to note that the displacements in `sdisp`/`sdisp_raw` should be defined as in the corresponding base collective.
- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. The `src_blk` and `src_blk_raw` arguments, where present, represent a limit for the number of elements in each private source array, and the block phases in `sdisp`/`sdisp_raw` are defined analogously to the *get* variant.
- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, the selected elements from array `A` are gathered in the shared memory block associated to thread 1 in array `B`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
```

```
shared [NELEMS] int B[NELEMS*THREADS];
shared int sdisp[THREADS];
shared int nelems[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}
sdisp[MYTHREAD] = MYTHREAD*NELEMS + (MYTHREAD % (NELEMS/2));
nelems[MYTHREAD] = MYTHREAD % (NELEMS/2) + 1;

upc_all_gather_v(&B[NELEMS+1], A, sdisp, NELEMS, NELEMS, nelems, sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.4. The upc_all_gather_all_v Collective

Synopsis

```
#include <upc.h>
void upc_all_gather_all_v (
         shared void *dst, shared const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_local (
         shared void *dst, shared const void *src, shared int *ddisp_local,
         shared int *sdisp_local, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_privparam (
         shared void *dst, shared const void *src, int ddisp,
         int *sdisp, size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_raw (
         shared void *dst, shared const void *src, shared size_t *ddisp_raw,
         shared size_t *sdisp_raw, shared size_t *nbytes, size_t dst_blk_raw,
         size_t src_blk_raw, upc_flag_t sync_mode
);
/**
```

```
 * GET VERSIONS
 */
void upc_all_gather_all_v_get (
         void *dst, shared const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_local_get (
         void *dst, shared const void *src, shared int *ddisp_local,
         shared int *sdisp_local, shared size_t *nelems,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_privparam_get (
         void *dst, shared const void *src, int ddisp,
         int *sdisp, size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_raw_get (
         void *dst, shared const void *src, shared size_t *ddisp_raw,
         shared size_t *sdisp_raw, shared size_t *nbytes, size_t dst_blk_raw,
         size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * PUT VERSIONS
 */
void upc_all_gather_all_v_put (
         shared void *dst, const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_local_put (
         shared void *dst, const void *src, shared int *ddisp_local,
         shared int *sdisp_local, shared size_t *nelems, size_t dst_blk,
         size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_privparam_put (
         shared void *dst, const void *src, int ddisp,
         int *sdisp, size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_raw_put (
```

```
        shared void *dst, const void *src, shared size_t *ddisp_raw,
        shared size_t *sdisp_raw, shared size_t *nbytes, size_t dst_blk_raw,
        size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * PRIV VERSIONS
 */
void upc_all_gather_all_v_priv (
        void *dst, const void *src, shared int *ddisp,
        shared int *sdisp, shared size_t *nelems,
        size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_local_priv (
        void *dst, const void *src, shared int *ddisp_local,
        shared int *sdisp_local, shared size_t *nelems,
        size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_privparam_priv (
        void *dst, const void *src, int ddisp,
        int *sdisp, size_t *nelems,
        size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_gather_all_v_raw_priv (
        void *dst, const void *src, shared size_t *ddisp_raw,
        shared size_t *sdisp_raw, shared size_t *nbytes,
        size_t src_blk_raw, upc_flag_t sync_mode
);
```

Description

- This is a UPC allgather collective that uses a configurable amount of source and destination data per thread.
- The `upc_all_gather_all_v` function copies `THREADS` chunks of shared memory from each of the array indexes in `src` contained in `sdisp` to the shared memory spaces with affinity to each thread in array `dst` specified by the array indexes in `ddisp`. Each index in `sdisp` determines a source position for a chunk of `nelems[i]` elements of size `typesize` for each thread `i`, which are merged consecutively in index `ddisp[i]` of array `dst`. The number of elements in a block in arrays `src` and `dst` is `src_blk` and `dst_blk`, respectively. All indexes in `sdisp` and `ddisp` are absolute

array indexes. All values in array `nelems` must be strictly greater than zero.

- The `upc_all_gather_all_v_local` function is analogous to the previous one, but here the `ddisp_local` and `sdisp_local` arrays should contain block phases to each thread instead of absolute array indexes in order to indicate a value for a thread. For example, if `sdisp_local[i]` and `ddisp_local[i]` are 1 in a call to this collective, it will have the same behavior as with a call to `upc_all_gather_all_v` with `sdisp[i]` and `ddisp[i]` equal to `i*src_blk+1` and `i*dst_blk+1`, respectively.

- The `upc_all_gather_all_v_privparam` function defines `ddisp`, `sdisp` and `nelems` as private arrays in order to optimize its performance. Additionally, `ddisp` is here defined as a single integer, and its value may be different for each thread calling this collective.

- The `upc_all_gather_all_v_raw` function indicates the amount of data for communications on each thread as a chunk of bytes following the sizes indicated by array `nbytes`. The displacements in the source and destination arrays are consequently described in terms of bytes in parameters `sdisp_raw` and `ddisp_raw`, respectively, as well as in the block size parameters `src_blk_raw` and `dst_blk_raw`.

- The number of gathered elements per thread must not be higher than the difference of `(MYTHREAD+1)*dst_blk` and `ddisp[i]` for thread `i` (`(MYTHREAD+1)*dst_blk_raw` and `ddisp_raw[i]` for the `raw` function). Analogously, the source position for thread `i` must be less than or equal to the difference of `dst_blk` and `nelems[i]`. If any of these conditions is not fulfilled, the chunks received in array `dst` may be truncated.

- All these functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [src_blk*typesize] char [src_blk*typesize*THREADS]
  ```

  except the `upc_all_gather_all_v_raw` function, which treats `src` as a pointer to a shared memory area with type:

  ```
  shared [src_blk_raw] char [src_blk_raw*THREADS]
  ```

- The `dst` pointer in all functions is interpreted as a pointer to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

and the same applies to `upc_all_gather_all_v_raw`, with `dst_blk*typesize = dst_blk_raw`.

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective, and only the `local` function does not require the argument `dst_blk`/`dst_blk_raw`.
- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The `src_blk` and `src_blk_raw` arguments, where present, represent a limit for the number of elements in each private source array. It is important to note that the displacements in `sdisp`/`sdisp_raw` and `ddisp`/`ddisp_raw` should be defined as in the corresponding base collective.
- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. The `src_blk` and `src_blk_raw` arguments, where present, represent a limit for the number of elements in each private source array, and the block phases in `sdisp`/`sdisp_raw` and `ddisp`/`ddisp_raw` are defined analogously to the *get* variant.
- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, the selected elements from array `A` are gathered in all shared memory blocks of array `B`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
shared int sdisp[THREADS];
shared int ddisp[THREADS];
```

```
shared int nelems[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}
sdisp[MYTHREAD] = MYTHREAD*NELEMS + (MYTHREAD % (NELEMS/2));
// All threads gather the results in the second position of their array block
ddisp[MYTHREAD] = MYTHREAD*NELEMS + 1;
nelems[MYTHREAD] = MYTHREAD % (NELEMS/2) + 1;

upc_all_gather_all_v(B, A, ddisp, sdisp, NELEMS, NELEMS, nelems, sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.5. The upc_all_exchange_v Collective

Synopsis

```
#include <upc.h>
void upc_all_exchange_v (
         shared void *dst, shared const void *src, shared int *exchange,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_local (
         shared void *dst, shared const void *src, shared int *exchange_local,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_merge (
         shared void *dst, shared const void *src, shared int *exchange,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         shared int *disp, upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_local (
         shared void *dst, shared const void *src, shared int *exchange_local,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         shared int *disp_local, upc_flag_t sync_mode
);
void upc_all_exchange_v_privparam (
```

```
        shared void *dst, shared const void *src, int *exchange,
        size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_exchange_v_raw (
        shared void *dst, shared const void *src, shared size_t *exchange_raw,
        shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
        upc_flag_t sync_mode
);
/**
 * GET VERSIONS
 */
void upc_all_exchange_v_get (
        void *dst, shared const void *src, shared int *exchange,
        shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_exchange_v_local_get (
        void *dst, shared const void *src, shared int *exchange_local,
        shared size_t *nelems, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_get (
        void *dst, shared const void *src, shared int *exchange,
        shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        shared int *disp, upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_local_get (
        void *dst, shared const void *src, shared int *exchange_local,
        shared size_t *nelems, size_t src_blk, size_t typesize,
        shared int *disp_local, upc_flag_t sync_mode
);
void upc_all_exchange_v_privparam_get (
        void *dst, shared const void *src, int *exchange,
        size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
        upc_flag_t sync_mode
);
void upc_all_exchange_v_raw_get (
        void *dst, shared const void *src, shared size_t *exchange_raw,
        shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
        upc_flag_t sync_mode
```

```
);
/**
 * PUT VERSIONS
 */
void upc_all_exchange_v_put (
         shared void *dst, const void *src, shared int *exchange,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_local_put (
         shared void *dst, const void *src, shared int *exchange_local,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_put (
         shared void *dst, const void *src, shared int *exchange,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         shared int *disp, upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_local_put (
         shared void *dst, const void *src, shared int *exchange_local,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         shared int *disp_local, upc_flag_t sync_mode
);
void upc_all_exchange_v_privparam_put (
         shared void *dst, const void *src, int *exchange,
         size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_raw_put (
         shared void *dst, const void *src, shared size_t *exchange_raw,
         shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
         upc_flag_t sync_mode
);
/**
 * PRIV VERSIONS
 */
void upc_all_exchange_v_priv (
         void *dst, const void *src, shared int *exchange,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
```

```
);
void upc_all_exchange_v_local_priv (
         void *dst, const void *src, shared int *exchange_local,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_priv (
         void *dst, const void *src, shared int *exchange,
         shared size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         shared int *disp, upc_flag_t sync_mode
);
void upc_all_exchange_v_merge_local_priv (
         void *dst, const void *src, shared int *exchange_local,
         shared size_t *nelems, size_t dst_blk, size_t typesize,
         shared int *disp_local, upc_flag_t sync_mode
);
void upc_all_exchange_v_privparam_priv (
         void *dst, const void *src, int *exchange,
         size_t *nelems, size_t dst_blk, size_t src_blk, size_t typesize,
         upc_flag_t sync_mode
);
void upc_all_exchange_v_raw_priv (
         void *dst, const void *src, shared size_t *exchange_raw,
         shared size_t *nbytes, size_t dst_blk_raw, size_t src_blk_raw,
         upc_flag_t sync_mode
);
```

Description

- This is a UPC exchange collective that uses a configurable amount of source and destination data per thread.
- The `upc_all_exchange_v` function swaps `THREADS` chunks of shared memory from array `src` into array `dst` according to the array indexes included in `exchange`. Array `exchange` must have `THREADS*THREADS` elements. Locations `i*THREADS` to `(i+1)*THREADS-1` in `exchange` define the array indexes for source chunks in `src` and also the destination indexes for received chunks in `dst` used by thread `i`. For each index `i` in `exchange`, there is a `nelems[i]` that defines the number of elements in the chunk. The number of elements in a block in arrays `src` and `dst` is `src_blk` and `dst_blk`, respectively. All indexes in `exchange` are absolute array indexes. All

values in array `nelems` must be strictly greater than zero.

- The `upc_all_exchange_v_local` function is analogous to `upc_all_exchange_v`, but here the `exchange_local` array should contain block phases to each thread instead of absolute array indexes in order to indicate a value for a thread. For example, if the value of `exchange_local[i]` for thread `j` is 1 in a call to `upc_all_exchange_v_local`, the function will reference the same chunk as with a call to `upc_all_exchange_v` with `exchange[i]` equal to `j*src_blk+1`. Analogously, the corresponding remote chunk will be copied to the position `j*dst_blk+1` in the destination array.

- The `upc_all_exchange_v_merge` function is similar to `upc_all_exchange_v`, but it does not copy each memory chunk to the destination array according to the `exchange` array; instead, all chunks with affinity to a thread in the destination array are copied one after the other (that is, the chunk received from thread 1 is placed immediately after the last element of the chunk received from thread 0, and so on). In order to provide more flexibility, this collective takes an additional argument: an array of `THREADS` called `disp`, which indicates the position in the destination array where each thread should place the first element of the chunk received from thread 0. If `NULL` is passed as a value for `disp`, no displacement is applied in the destination array.

- The `upc_all_exchange_v_merge_local` function is analogous to the previous one, but here the arrays `exchange_local` and `disp_local` contain block phases to each thread instead of absolute array indexes in order to indicate a value for a thread. For example, if the values of `exchange_local[i]` and `disp_local[j]` for thread `j` are both 1 in a call to `upc_all_exchange_v_merge_local`, the function will reference the same chunk as with a call to `upc_all_exchange_v_merge` with `exchange[i]` equal to `j*src_blk+1`. Analogously, the corresponding remote chunks will be stored consecutively starting in the position `j*dst_blk+1` in the destination array.

- The `upc_all_exchange_v_privparam` function defines `exchange` and `nelems` as private arrays in order to optimize its performance.

- The `upc_all_exchange_v_raw` function indicates the amount of data for communications on each thread as a chunk of bytes following the sizes indicated by array `nbytes`. The displacements in the source and destination arrays are consequently described in terms of bytes in the parameter `exchange_raw`, as well as in the block size parameters `src_blk_raw` and `dst_blk_raw`.

- None of the chunks defined should exceed the size of its corresponding destination; that is, for each thread `i`, `nelems[i*THREADS+j]` should be less than or equal to the difference of `exchange[j*THREADS+i+1]` (or `dst_blk`*`THREADS` when `i=j=THREADS-1`) and `exchange[j*THREADS+i]`. The same applies to the `raw` collectives considering `nbytes`, `dst_blk_raw` and `exchange_raw` in the previous cases. If these conditions are not fulfilled, the chunks received in array `dst` may be truncated.

- All these functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [src_blk*typesize] char [src_blk*typesize*THREADS]
  ```

  except `upc_all_exchange_v_raw`, which treats `src` as a pointer to a shared memory area with type:

  ```
  shared [src_blk_raw] char [src_blk_raw*THREADS]
  ```

- The `dst` pointer in all functions is interpreted as a pointer to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  considering `dst_blk*typesize = dst_blk_raw` for `upc_all_exchange_v_raw`.

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective. The `dst_blk`/`dst_blk_raw` argument, where present, represents a limit for the number of elements in each private destination array. It is important to note that if non-local values are used in `exchange`/`exchange_raw`, each value should be defined as if the destination array was a shared array with the same type as the corresponding base collective.

- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The `src_blk`/`src_blk_raw` argument, where present, represents a limit for the number of elements in each private source array. Similarly to the `_get` case, when using non-local values in `exchange`/`exchange_raw` each value should be defined as if the source array was a shared array with the same type as the corresponding base collective.

- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. The same comments as in the `_put` case apply here.

- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, the selected elements from array `A` are exchanged in array `B` (assume `THREADS` = `NELEMS/2`).

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
shared [THREADS] exchange[THREADS];
shared [THREADS] int nelems[THREADS*THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

// Initialize 'exchange' array
for (int i=MYTHREAD*THREADS; i<(MYTHREAD+1)*THREADS; i++) {
    exchange[i]=i+1;
}
// Move always one or two elements per chunk
for (int i=MYTHREAD*THREADS; i<(MYTHREAD+1)*THREADS; i++) {
    nelems[i]=1+(i%2);
```

```
}

upc_all_exchange_v(B, A, exchange, nelems, NELEMS, NELEMS, sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.6. The upc_all_permute_v Collective

Synopsis

```
#include <upc.h>
void upc_all_permute_v (
         shared void *dst, shared const void *src, shared const int *perm,
         shared int *disp, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_local (
         shared void *dst, shared const void *src, shared const int *perm,
         shared int *disp_local, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_raw (
         shared void *dst, shared const void *src, shared const int *perm,
         shared size_t *disp_raw, shared size_t *nbytes, size_t dst_blk_raw,
         size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * GET VERSIONS
 */
void upc_all_permute_v_get (
         void *dst, shared const void *src, shared const int *perm,
         shared int *disp, shared size_t *nelems, size_t dst_blk,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_local_get (
         void *dst, shared const void *src, shared const int *perm,
         shared int *disp_local, shared size_t *nelems,
         size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_raw_get (
         void *dst, shared const void *src, shared const int *perm,
```

```
        shared size_t *disp_raw, shared size_t *nbytes, size_t dst_blk_raw,
        size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * PUT VERSIONS
 */
void upc_all_permute_v_put (
        shared void *dst, const void *src, shared const int *perm,
        shared int *disp, shared size_t *nelems, size_t dst_blk,
        size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_local_put (
        shared void *dst, const void *src, shared const int *perm,
        shared int *disp_local, shared size_t *nelems, size_t dst_blk,
        size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_raw_put (
        shared void *dst, const void *src, shared const int *perm,
        shared size_t *disp_raw, shared size_t *nbytes, size_t dst_blk_raw,
        size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * PRIV VERSIONS
 */
void upc_all_permute_v_priv (
        void *dst, const void *src, shared const int *perm,
        shared int *disp, shared size_t *nelems, size_t dst_blk,
        size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_local_priv (
        void *dst, const void *src, shared const int *perm,
        shared int *disp_local, shared size_t *nelems, size_t dst_blk,
        size_t typesize, upc_flag_t sync_mode
);
void upc_all_permute_v_raw_priv (
        void *dst, const void *src, shared const int *perm,
        shared size_t *disp_raw, shared size_t *nbytes, size_t dst_blk_raw,
        size_t src_blk_raw, upc_flag_t sync_mode
);
```

Description

- This is a UPC permute collective that uses a configurable amount of source and destination data per thread.
- The `upc_all_permute_v` function copies a chunk of shared memory with affinity to the `i`th thread to a portion of a shared memory area associated to thread `perm[i]` in the same array. The start index of each of the `THREADS` chunks is stored in `disp`, and the size of each of them is indicated by the corresponding element in array `nelems`. All elements in `disp` are absolute array indexes. All elements in `nelems` must be strictly greater than zero. The number of elements per block in arrays `src` and `dst` is `src_blk` and `dst_blk`, respectively.
- The `upc_all_permute_v_local` function is analogous to `upc_all_exchange_v`, but here the `disp_local` array should contain block phases to each thread instead of absolute array indexes in order to indicate a value for a thread. For example, if the value of `disp_local[i]` is 1 in a call to `upc_all_permute_v_local`, the destination array will be updated as with a call to `upc_all_permute_v` with `disp[i]` equal to `i*dst_blk+1`.
- The `upc_all_permute_v_raw` function indicates the amount of data for communications on each thread as a chunk of bytes following the sizes indicated by array `nbytes`. The displacements in the source and destination arrays are consequently described in terms of bytes in the parameter `ddisp_raw`, as well as in the block size parameters `src_blk_raw` and `dst_blk_raw`.
- The target of the `src`, `dst` and `perm` pointers must have affinity to thread 0. Additionally, the `src` and `dst` pointers are treated as if they had phase 0.
- The destination position for thread `i` must be less than or equal to the difference of `dst_blk`/`dst_blk_raw` and `nelems[i]`/`nbytes[i]`. If any of these conditions is not fulfilled, the behavior of the function may be undefined or the chunks received in array `dst` may be truncated.
- The values stored in `perm[0...THREADS-1]` must define a permutation of the integer values between `0` and `THREADS-1`.
- All these functions treat the `src` pointer as if it pointed to a shared memory area with type:

```
shared [src_blk*typesize] char [src_blk*typesize*THREADS]
```

except `upc_all_permute_v_raw`, which treats `src` as a pointer to a shared memory area with type:

```
shared [src_blk_raw] char [src_blk_raw*THREADS]
```

- The `dst` pointer in all functions is interpreted as a pointer to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  considering `dst_blk*typesize = dst_blk_raw` for `upc_all_permute_v_raw`.

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective. The `dst_blk`/`dst_blk_raw` argument, where present, represents a limit for the number of elements in each private destination array.
- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The `src_blk`/`src_blk_raw` argument, where present, represents a limit for the number of elements in each private source array.
- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. The `src_blk`/`src_blk_raw` argument, where present, represents a limit for the number of elements in each private source array.
- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, the shared memory blocks in array `A` are permuted into array `B` according to the definitions in `perm`. This code is specifically designed for 4 threads.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
shared int perm[THREADS];
shared int disp[THREADS];
shared int nelems[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}
if (MYTHREAD == 0) {
    // Define the desired permutation
    perm[0]=3;   perm[1]=2;
    perm[2]=0;   perm[3]=1;
    // Four 2-element chunks are defined
    for (int i=0; i<THREADS; i++) {
        disp[i]=i*2;
        nelems[i]=2;
    }
}

upc_all_permute_v(B, A, disp, nelems, perm, NELEMS, NELEMS, sizeof(int),
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.7. The upc_all_vector_copy Collective

Synopsis

```
#include <upc.h>
void upc_all_vector_copy (
         shared void *dst, shared const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_privparam (
         shared void *dst, shared const void *src, int *ddisp,
         int *sdisp, size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
```

```
void upc_all_vector_copy_raw (
         shared void *dst, shared const void *src, shared size_t *ddisp_raw,
         shared size_t *sdisp_raw, shared size_t *nbytes, int nchunks,
         size_t dst_blk_raw, size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * GET VERSIONS
 */
void upc_all_vector_copy_get (
         void *dst, shared const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_privparam_get (
         void *dst, shared const void *src, int *ddisp,
         int *sdisp, size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_raw_get (
         void *dst, shared const void *src, shared size_t *ddisp_raw,
         shared size_t *sdisp_raw, shared size_t *nbytes, int nchunks,
         size_t dst_blk_raw, size_t src_blk_raw, upc_flag_t sync_mode
);
/**
 * PUT VERSIONS
 */
void upc_all_vector_copy_put (
         shared void *dst, const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_privparam_put (
         shared void *dst, const void *src, int *ddisp,
         int *sdisp, size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_raw_put (
         shared void *dst, const void *src, shared size_t *ddisp_raw,
         shared size_t *sdisp_raw, shared size_t *nbytes, int nchunks,
         size_t dst_blk_raw, size_t src_blk_raw, upc_flag_t sync_mode
);
```

```
/**
 * PRIV VERSIONS
 */
void upc_all_vector_copy_priv (
         void *dst, const void *src, shared int *ddisp,
         shared int *sdisp, shared size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_privparam_priv (
         void *dst, const void *src, int *ddisp,
         int *sdisp, size_t *nelems, int nchunks,
         size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_vector_copy_raw_priv (
         void *dst, const void *src, shared size_t *ddisp_raw,
         shared size_t *sdisp_raw, shared size_t *nbytes, int nchunks,
         size_t dst_blk_raw, size_t src_blk_raw, upc_flag_t sync_mode
);
```

Description

- This is a UPC collective that provides a flexible definition for any number of data copies between threads.

- The `upc_all_vector_copy` function copies `nchunks` portions of shared memory of array `src` into array `dst`. Arrays `sdisp`, `ddisp` and `nelems` must have `nchunks` elements, in order to define the array index of the source subarray, the destination index and the chunk length, respectively. All values in `nelems` must be strictly greater than zero. The number of elements per block in array `src` and `dst` is `src_blk` and `dst_blk`, respectively.

- The `upc_all_vector_copy_privparam` function defines `sdisp`, `ddisp` and `nelems` as private arrays in order to optimize its performance.

- The `upc_all_vector_copy_raw` function indicates the amount of data for communications on each thread as a chunk of bytes following the sizes indicated by array `nbytes`. The displacements in the source and destination arrays are consequently described in terms of bytes in parameters `sdisp_raw` and `ddisp_raw`, respectively, as well as in the block size parameters `src_blk_raw` and `dst_blk_raw`.

- All chunks defined must have affinity to only one thread; that is, for each chunk `i`, `nelems[i]` should be less than or equal to the difference of `dst_blk` and `ddisp[i]` (`nbytes[i]`, `dst_blk_raw` and `ddisp_raw[i]` for the `_raw` collective). Additionally, the target of the `src` and `dst` pointers must have affinity to thread 0. If any of these conditions is not fulfilled or if the chunks overlap, the chunks received in array `dst` may be truncated or the result of the copy may be undefined.

- All these functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [src_blk*typesize] char [src_blk*typesize*THREADS]
  ```

  except `upc_all_vector_copy_raw`, which treats `src` as a pointer to a shared memory area with type:

  ```
  shared [src_blk_raw] char [src_blk_raw*THREADS]
  ```

- The `dst` pointer in all functions is interpreted as a pointer to a shared memory area with type:

  ```
  shared [dst_blk*typesize] char [dst_blk*typesize*THREADS]
  ```

  considering `dst_blk*typesize = dst_blk_raw` for `upc_all_vector_copy_raw`.

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective. Here the `dst_blk`/`dst_blk_raw` argument represents a limit for the number of elements in each private destination array.

- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. Here the `src_blk`/`src_blk_raw` argument represents a limit for the number of elements in each private source array.

- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis. Here the `src_blk`/`src_blk_raw` argument represents a limit for the number of elements in each private source array.

- Apart from the commented parameters, all the arguments in common with the base collectives are analogously defined.

Example

In the next piece of code, four chunks of shared memory are defined in array `A` according to the displacements in `sdisp` and the number of elements in `nelems`, and they are moved to array `B` in the positions stated in `ddisp`.

```
#define NELEMS 1000
#define NCHUNKS 4
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
shared int sdisp[NCHUNKS];
shared int ddisp[NCHUNKS];
shared int nelems[NCHUNKS];

// Initialize A
for (int i=MYTHREAD*NELEMS*THREADS; i<(MYTHREAD+1)*NELEMS*THREADS; i++) {
    A[i]=i;
}

if (MYTHREAD == 0) {
    // Four 110-element chunks are defined
    for (int i=0; i<NCHUNKS; i++) {
        sdisp[i]=i*100+15;
        ddisp[i]=i*200;
        nelems[i]=110;
    }
}

upc_all_vector_copy(B, A, ddisp, sdisp, nelems, NCHUNKS, NELEMS, NELEMS,
    sizeof(int), UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.2.8. Computational Vector-variant Collectives

Synopsis

```
#include <upc.h>
#include <upc_collective.h>
/**
 * REDUCE
 */
void upc_all_reduce<<T>>_v (
         shared void *dst, shared const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_v_get (
         void *dst, shared const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_v_put (
         shared void *dst, const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_v_priv (
         void *dst, const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
/**
 * ALLREDUCE
 */
void upc_all_reduce<<T>>_all_v (
         shared void *dst, shared const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_v_get (
         void *dst, shared const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
```

```
);
void upc_all_reduce<<T>>_all_v_put (
         shared void *dst, const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_v_priv (
         void *dst, const void *src, upc_op_t op,
         shared int *sdisp, shared size_t *nelems, int nchunks, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
/**
 * PREFIX REDUCE
 */
void upc_all_prefix_reduce_<<T>>_v (
         shared void *dst, shared const void *src, upc_op_t op,
         shared int *ddisp, shared int *sdisp, shared size_t *nelems,
         int nchunks, size_t dst_blk, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_prefix_reduce_<<T>>_v_get (
         void *dst, shared const void *src, upc_op_t op,
         shared int *ddisp, shared int *sdisp, shared size_t *nelems,
         int nchunks, size_t dst_blk, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_prefix_reduce_<<T>>_v_put (
         shared void *dst, const void *src, upc_op_t op,
         shared int *ddisp, shared int *sdisp, shared size_t *nelems,
         int nchunks, size_t dst_blk, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
void upc_all_prefix_reduce_<<T>>_v_priv (
         void *dst, const void *src, upc_op_t op,
         shared int *ddisp, shared int *sdisp, shared size_t *nelems,
         int nchunks, size_t dst_blk, size_t src_blk,
         <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), upc_flag_t sync_mode
);
```

Description

- These functions define a set of UPC computational operations that provide a flexible definition for any number of source data chunks.
- The prototypes above represent 132 computational extended collectives, where *T* and *TYPE* have the following correspondences:

T	*TYPE*	*T*	*TYPE*
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

For example, if *T* is C, *TYPE* must be signed char

- The upc_all_reduce*T*_v and upc_all_reduce*T*_all_v functions reduce a subset of values in array src according to a custom number of chunks defined by the additional parameters: sdisp contains the array indexes for every chunk, nelems indicates the length of each chunk and nchunks represents the number of chunks that are used in each function call. Therefore, arrays sdisp and nelems should have nchunks elements each. The src_blk parameter represents the number of elements included in a block of memory in the src array. Additionally to this, the upc_all_reduce*T*_all_v function returns the result in the shared memory space of all threads.
- The upc_all_prefix_reduce*T*_v function computes a prefix reduction using a custom number of chunks defined in array src. The parameter sdisp contains the array indexes for every chunk, ddisp defines the destination indexes for each processed chunk, nelems indicates the length of each chunk and nchunks represents the number of chunks that are used in each function call. Therefore, arrays sdisp, ddisp and nelems should have nchunks elements each. After the execution of the function, the dst array contains the results of the partial reductions obtained for the defined chunks according to the ordering of the source values in src, and according to the block size definitions src_blk and dst_blk (if they are not equal, the result may be truncated).
- On completion of the upc_all_reduce*T*_v functions, the *TYPE* value referenced by dst is src[<$chunk_0$>] $\oplus$ src[<$chunk_1$>] $\oplus \cdots \oplus$ src[<$chunk_{nchunks-1}$>], where

`src[<`$chunk_n$`>]` is the result of the reduction of all the elements in chunk `n` and "$\oplus$" is the operator specified by the `op` parameter.

- On completion of the `upc_all_reduce`*`T`*`_all_v` functions, the *`TYPE`* value referenced by `&(dst + i)` for each thread `i` is `src[<`$chunk_0$`>]` $\oplus$ `src[<`$chunk_1$`>]` $\oplus \cdots \oplus$ `src[<`$chunk_{nchunks-1}$`>]`, where `src[<`$chunk_n$`>]` is the result of the reduction of all the elements in chunk `n` and "$\oplus$" is the operator specified by the `op` parameter. It is important to note that the operation is performed regardless of the block size in array `dst`.

- On completion of the `upc_all_prefix_reduce`*`T`*`_v` functions, the *`TYPE`* value referenced by the `i`th position of chunk `j` is $\bigoplus_{n=0}^{j-1}$`src[<`$chunk_n$`>]` $\oplus$ `src[<`$chunk_{j(0)}$`>]` $\oplus$ `src[<`$chunk_{j(1)}$`>]` $\oplus \cdots \oplus$ `src[<`$chunk_{j(i)}$`>]`, where $\bigoplus_{n=0}^{j-1}$`src[<`$chunk_n$`>]` is the result of the reduction of all the elements in chunks `0` to `j-1`, `src[<`$chunk_{j(i)}$`>]` is the `i`th element of chunk `j` and "$\oplus$" is the operator specified by the `op` parameter.

- The argument `op` can have the following values:

  - `UPC_ADD`: addition.
  - `UPC_MULT`: multiplication.
  - `UPC_AND`: bitwise `AND` for integer and character variables. Results are undefined for floating point numbers.
  - `UPC_OR`: bitwise `OR` for integer and character variables. Results are undefined for floating point numbers.
  - `UPC_XOR`: bitwise `XOR` for integer and character variables. Results are undefined for floating point numbers.
  - `UPC_LOGAND`: logical `AND` for all variable types.
  - `UPC_LOGOR`: logical `OR` for all variable types.
  - `UPC_MIN`: for all data types, find the minimum value.
  - `UPC_MAX`: for all data types, find the maximum value.
  - `UPC_FUNC`: use the specified commutative function `func` to operate on the data in the `src` array.
  - `UPC_NONCOMM_FUNC`: use the specified non-commutative function `func` to operate on the data in the `src` array.

- The operations represented by `op` are assumed to be associative and commutative (except those provided using `UPC_NONCOMM_FUNC`). An operation whose result is dependent on the operator evaluation or on the order of the operands will have undefined results.
- If the value of `src_blk` passed to all these functions is greater than zero, they treat the `src` pointer as if it pointed to a shared memory area with blocking factor `src_blk`, and therefore with type:

  ```
  shared [src_blk] <<TYPE>> [<nelems>]
  ```

  where `<nelems>` represents the total number of elements in the array. It is important to note that this value is not explicitly passed as a parameter to the functions.
- If the value of `src_blk` is zero, the functions treat the `src` pointer as if it pointed to a shared memory area of `<nelems>` elements of type *`TYPE`* with an indefinite layout qualifier, and therefore with type:

  ```
  shared [] <<TYPE>> [<nelems>]
  ```

  In this case, the source array would only have affinity to one thread, and thus the chunks for reductions should be defined accordingly.
- The `upc_all_reduce`*`T`*`_v` and `upc_all_reduce`*`T`*`_all_v` functions consider the `dst` array as if it had type:

  ```
  shared <<TYPE>> *
  ```

- The `upc_all_prefix_reduce`*`T`*`_v` function considers the `dst` array as if it had type:

  ```
  shared [dst_blk] <<TYPE>> [<nelems>]
  ```

Get-put-priv Variants

- A variant called *get* is defined by using a private pointer as destination for communications. The name of this variant is the same as the base collective, but adding `_get` at the end, as shown in the synopsis. The shared source pointers in each of these functions are treated the same way as in the corresponding base collective.

- A variant called *put* is defined by using a private pointer as source for communications. The name of this variant is the same as the base collective, but adding `_put` at the end, as shown in the synopsis. The shared destination pointers in each of these functions are treated the same way as in the corresponding base collective.
- A variant called *priv* is defined by using private pointers as source and destination for communications. The name of this variant is the same as the base collective, but adding `_priv` at the end, as shown in the synopsis.
- In all these variants the `src_blk` argument represents a limit for the number of elements of the source chunk. Even when `src` is private, the displacements in `sdisp` and `ddisp` (when present) should be defined as if `src` had the same shared type as in the corresponding base collective.
- Apart from the commented parameters, all of these variants present the same additional arguments as the base collective.

Examples

In the next piece of code, the sum of 4 chunks in array `A` is computed and stored in the shared memory of thread 0. Then, the sum of the first three chunks is computed and stored in the shared memory of all threads. Finally, the prefix reduction is applied to the first two chunks and stored in array `B`. It is important to note that, in all cases, it is not relevant the actual number of elements in `sdisp`, `ddisp` and `nelems`, as long as they have at least the required number of chunks for the collective call in which they are used.

```
#define NELEMS 1000
#define NCHUNKS 4
shared [THREADS] int A[NELEMS*THREADS];
shared [THREADS] int B[NELEMS*THREADS];
shared int sdisp[NCHUNKS], ddisp[NCHUNKS], nelems[NCHUNKS];
shared int res, result[THREADS];

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

if (MYTHREAD == 0) {
```

```
    // Four 110-element chunks are defined
    for (int i=0; i<NCHUNKS; i++) {
        sdisp[i]=i*100+15;   ddisp[i]=i*100+15;   nelems[i]=110;
    }
}

upc_all_reduceI_v(&res, A, UPC_ADD, sdisp, nelems, NCHUNKS,
    THREADS, NULL, UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_reduceI_all_v(result, A, UPC_ADD, sdisp, nelems, NCHUNKS-1,
    THREADS, NULL, UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_prefix_reduceI_v(B, A, UPC_ADD, ddisp, sdisp, nelems, NCHUNKS-2,
    THREADS, THREADS, NULL, UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3. Team-based Collectives

A.3.1. The upc_all_broadcast_team Collective

Synopsis

```
#include <upc.h>
#include "teamutil.h"
void upc_all_broadcast_team (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_broadcast_team_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_broadcast_team_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_broadcast_team_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
```

Description

- This is a UPC broadcast collective that is executed by a subset of threads defined by a team (parameter `t`).
- Each memory chunk involved in the communication has a size of `nbytes/nthr_team`, where `nthr_team` is the number of threads in the team. The rest of arguments are similar to the corresponding standard collective.
- The `upc_all_broadcast_team` function treats the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [] char [nbytes]
  ```

  and the `dst` pointer is considered as if it pointed to a shared memory area with type:

  ```
  shared [nbytes] char [nbytes*THREADS]
  ```

Get-put-priv Variants

Three variants are defined using different configurations for the locality of the source and destination pointers. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the base collective according to the previous description.

Example

In the next piece of code, the whole shared memory block associated to thread 1 in array `A` is copied to the blocks in array `B` associated to all threads in the team.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
team t;   // The team has to be initialized

// Initialize A
```

```
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_broadcast_team(B, &A[NELEMS], NELEMS*sizeof(int), t,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3.2. The upc_all_scatter_team Collective

Synopsis

```
#include <upc.h>
#include "teamutil.h"
/**
 * TEAM
 */
void upc_all_scatter_team (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_scatter_team_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_scatter_team_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_scatter_team_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
/**
 * TEAM ALLTHR
 */
void upc_all_scatter_team_allthr (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_scatter_team_allthr_get (
```

```
        void *dst, shared const void *src, size_t nbytes, team t,
        upc_flag_t sync_mode
);
void upc_all_scatter_team_allthr_put (
        shared void *dst, const void *src, size_t nbytes, team t,
        upc_flag_t sync_mode
);
void upc_all_scatter_team_allthr_priv (
        void *dst, const void *src, size_t nbytes, team t,
        upc_flag_t sync_mode
);
```

Description

- This is a UPC scatter collective that is executed by a subset of threads defined by a team (parameter `t`).
- In the `upc_all_scatter_team` collective each memory chunk involved in the communication has a size of `nbytes/nthr_team`, where `nthr_team` is the number of threads in the team. The rest of arguments are similar to the corresponding standard collective.
- The `upc_all_scatter_team_allthr` collective is analogous to the previous one, but every chunk has a size of `nbytes/THREADS`.
- Both functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [] char [nbytes]
  ```

  and the `dst` pointer is considered as if it pointed to a shared memory area with type:

  ```
  shared [nbytes] char [nbytes*THREADS]
  ```

Get-put-priv Variants

Three variants are defined for each of the two functions presented before. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as

source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the corresponding base collective according to the previous description.

Example

In the next piece of code, the shared memory block associated to thread 1 in array `A` is scattered to array `B`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
team t;   // The team has to be initialized

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_scatter_team(B, &A[NELEMS], NELEMS*sizeof(int), t,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3.3. The `upc_all_gather_team` Collective

Synopsis

```
#include <upc.h>
#include "teamutil.h"
/**
 * TEAM
 */
void upc_all_gather_team (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_team_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
```

```
void upc_all_gather_team_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_team_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
/**
 * TEAM ALLTHR
 */
void upc_all_gather_team_allthr (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_team_allthr_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_team_allthr_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_team_allthr_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
```

Description

- This is a UPC gather collective that is executed by a subset of threads defined by a team (parameter `t`).
- In the `upc_all_gather_team` collective each memory chunk involved in the communication has a size of `nbytes`/`nthr_team`, where `nthr_team` is the number of threads in the team. The rest of arguments are similar to the corresponding standard collective.
- The `upc_all_gather_team_allthr` collective is analogous to the previous one, but every chunk has a size of `nbytes`/`THREADS`.

- Both functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [] char [nbytes]
  ```

  and the `dst` pointer is considered as if it pointed to a shared memory area with type:

  ```
  shared [nbytes] char [nbytes*THREADS]
  ```

Get-put-priv Variants

Three variants are defined for each of the two functions presented before. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the corresponding base collective according to the previous description.

Example

In the next piece of code, two integers per thread from array `A` are gathered in the memory block of array `B` with affinity to thread 1.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
team t;   // The team has to be initialized

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_gather_team(&B[NELEMS], A, 2*sizeof(int), t,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3.4. The upc_all_gather_all_team Collective

Synopsis

```
#include <upc.h>
#include "teamutil.h"
/**
 * TEAM
 */
void upc_all_gather_all_team (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_all_team_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_all_team_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_all_team_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
/**
 * TEAM ALLTHR
 */
void upc_all_gather_all_team_allthr (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_all_team_allthr_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_all_team_allthr_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_gather_all_team_allthr_priv (
```

```
        void *dst, const void *src, size_t nbytes, team t,
        upc_flag_t sync_mode
);
```

Description

- This is a UPC allgather collective that is executed by a subset of threads defined by a team (parameter `t`).
- In the `upc_all_gather_all_team` collective each memory chunk involved in the communication has a size of `nbytes/nthr_team`, where `nthr_team` is the number of threads in the team. The rest of arguments are similar to the corresponding standard collective.
- The `upc_all_gather_all_team_allthr` collective is analogous to the previous one, but every chunk has a size of `nbytes/THREADS`.
- Both functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes] char [nbytes*THREADS]
  ```

  and the `dst` pointer is considered as if it pointed to a shared memory area with type:

  ```
  shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]
  ```

Get-put-priv Variants

Three variants are defined for each of the two functions presented before. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the corresponding base collective according to the previous description.

Example

In the next piece of code, two integers per thread are gathered from array `A` to array `B` by all threads in team `t`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
team t;   // The team has to be initialized

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_gather_all_team(B, A, 2*sizeof(int), t, UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3.5. The upc_all_exchange_team Collective

Synopsis

```
#include <upc.h>
#include "teamutil.h"
/**
 * TEAM
 */
void upc_all_exchange_team (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_exchange_team_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_exchange_team_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_exchange_team_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
/**
 * TEAM ALLTHR
 */
```

```
void upc_all_exchange_team_allthr (
         shared void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_exchange_team_allthr_get (
         void *dst, shared const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_exchange_team_allthr_put (
         shared void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
void upc_all_exchange_team_allthr_priv (
         void *dst, const void *src, size_t nbytes, team t,
         upc_flag_t sync_mode
);
```

Description

- This is a UPC exchange collective that is executed by a subset of threads defined by a team (parameter `t`).
- In the `upc_all_exchange_team` collective each memory chunk involved in the communication has a size of `nbytes/nthr_team`, where `nthr_team` is the number of threads in the team. The rest of arguments are similar to the corresponding standard collective.
- The `upc_all_exchange_team_allthr` collective is analogous to the previous one, but every chunk has a size of `nbytes/THREADS`.
- Both functions treat the `src` pointer as if it pointed to a shared memory area with type:

  ```
  shared [nbytes] char [nbytes*THREADS]
  ```

  and the `dst` pointer is considered as if it pointed to a shared memory area with type:

  ```
  shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]
  ```

Get-put-priv Variants

Three variants are defined for each of the two functions presented before. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the corresponding base collective according to the previous description.

Example

In the next piece of code, two integers per thread are exchanged from array `A` to array `B` by all threads in team `t`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
team t;   // The team has to be initialized

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_exchange_team(B, A, 2*sizeof(int), t, UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3.6. The `upc_all_permute_team` Collective

Synopsis

```
#include <upc.h>
#include "teamutil.h"
/**
 * TEAM
 */
void upc_all_permute_team (
         shared void *dst, shared const void *src, shared const int *perm
         size_t nbytes, team t, upc_flag_t sync_mode
);
```

```
void upc_all_permute_team_get (
         void *dst, shared const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
void upc_all_permute_team_put (
         shared void *dst, const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
void upc_all_permute_team_priv (
         void *dst, const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
/**
 * TEAM ALLTHR
 */
void upc_all_permute_team_allthr (
         shared void *dst, shared const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
void upc_all_permute_team_allthr_get (
         void *dst, shared const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
void upc_all_permute_team_allthr_put (
         shared void *dst, const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
void upc_all_permute_team_allthr_priv (
         void *dst, const void *src, shared const int *perm,
         size_t nbytes, team t, upc_flag_t sync_mode
);
```

Description

- This is a UPC permute collective that is executed by a subset of threads defined by a team (parameter `t`).
- The values stored in `perm[0...nthr_team]` (where `nthr_team` is the number of threads in the team) must define a permutation using the integer values between `0` and `nthr_team-1`. The rest of arguments are similar to the corresponding standard

collective.

- The `upc_all_permute_team_allthr` collective is analogous to the previous one, but the permutation is defined using thread identifiers, and thus `perm` should have `THREADS` values.

- Both functions treat the `src` and `dst` pointers as if they pointed to a shared memory area with type:

```
shared [nbytes] char [nbytes*THREADS]
```

Get-put-priv Variants

Three variants are defined for each of the two functions presented before. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the corresponding base collective according to the previous description.

Example

In the next piece of code, the blocks associated to each thread in `A` are permuted in array `B` by all threads in team `t`.

```
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
shared int perm[THREADS];   // perm has to be initialized
team t;   // The team has to be initialized

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_permute_team(B, A, perm, NELEMS*sizeof(int), t,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.3.7. Computational Team-based Collectives

Synopsis

```
#include <upc.h>
#include <upc_collective.h>
#include "teamutil.h"
/**
 * REDUCE
 */
void upc_all_reduce<<T>>_team (
         shared void *dst, shared const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_team_get (
         void *dst, shared const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_team_put (
         shared void *dst, const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_team_priv (
         void *dst, const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
         upc_flag_t sync_mode
);
/**
 * ALLREDUCE
 */
void upc_all_reduce<<T>>_all_team (
         shared void *dst, shared const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_team_get (
         void *dst, shared const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
```

```
        upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_team_put (
        shared void *dst, const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
        upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_team_priv (
        void *dst, const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
        upc_flag_t sync_mode
);
/**
 * PREFIX REDUCE
 */
void upc_all_prefix_reduce<<T>>_team (
        shared void *dst, shared const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
        upc_flag_t sync_mode
);
void upc_all_prefix_reduce<<T>>_team_get (
        void *dst, shared const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
        upc_flag_t sync_mode
);
void upc_all_prefix_reduce<<T>>_team_put (
        shared void *dst, const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
        upc_flag_t sync_mode
);
void upc_all_prefix_reduce<<T>>_team_priv (
        void *dst, const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>), team t,
        upc_flag_t sync_mode
);
```

Description

- These functions define a set of UPC computational operations that are executed by a subset of threads defined by a team (parameter `t`).

- The prototypes above represent 132 computational extended collectives, where *T* and *TYPE* have the following correspondences:

T	*TYPE*	*T*	*TYPE*
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

For example, if *T* is C, *TYPE* must be signed char

- The arguments used in upc_all_reduce*T*_team and upc_all_reduce*T*_all_team are analogous to those of the standard reduce collective. This also applies to upc_all_prefix_reduce*T*_team with its standard collective counterpart.
- Both src and dst pointers must have affinity to a thread included in the team.
- The order of processing of the values in src is determined by the team identifier of each thread; that is, after processing thread i with team identifier m, the next chunk processed will correspond to thread j with team identifier m+1.
- On completion of the upc_all_reduce*T*_team functions, the value of the *TYPE* shared variable referenced by dst is src[0] $\oplus$ src[1] $\oplus \cdots \oplus$ src[nelems-1], where src[i] is the ith element in the reduction sequence and "$\oplus$" is the operator specified by the op parameter. The upc_all_reduce*T*_all_team function presents an analogous result, but here the result is stored in the addresses referenced by &(dst + m), where m is the identifier of each thread in the team.
- On completion of the upc_all_prefix_reduce*T*_team functions, the ith *TYPE* shared variable in the destination (dst[i]) is src[0] $\oplus$ src[1] $\oplus \cdots \oplus$ src[i].
- The argument op can have the following values:
  - UPC_ADD: addition.
  - UPC_MULT: multiplication.
  - UPC_AND: bitwise AND for integer and character variables. Results are undefined for floating point numbers.

- `UPC_OR`: bitwise `OR` for integer and character variables. Results are undefined for floating point numbers.
- `UPC_XOR`: bitwise `XOR` for integer and character variables. Results are undefined for floating point numbers.
- `UPC_LOGAND`: logical `AND` for all variable types.
- `UPC_LOGOR`: logical `OR` for all variable types.
- `UPC_MIN`: for all data types, find the minimum value.
- `UPC_MAX`: for all data types, find the maximum value.
- `UPC_FUNC`: use the specified commutative function `func` to operate on the data in the `src` array.
- `UPC_NONCOMM_FUNC`: use the specified non-commutative function `func` to operate on the data in the `src` array.

- The operations represented by `op` are assumed to be associative and commutative (except those provided using `UPC_NONCOMM_FUNC`). An operation whose result is dependent on the operator evaluation or on the order of the operands will have undefined results.
- If the value of `blk_size` passed to all these functions is greater than zero, they treat the `src` pointer as if it pointed to a shared memory area with blocking factor `blk_size`, and therefore with type:

  ```
  shared [blk_size] <<TYPE>> [<nelems>]
  ```

  where `<nelems>` represents the total number of elements in the array. It is important to note that this value is not explicitly passed as a parameter to the functions.
- If the value of `blk_size` is zero, the functions treat the `src` pointer as if it pointed to a shared memory area of `<nelems>` elements of type *`TYPE`* with an indefinite layout qualifier, and therefore with type:

  ```
  shared [] <<TYPE>> [<nelems>]
  ```

- The `upc_all_reduce`*`T`*`_team` and `upc_all_reduce`*`T`*`_all_team` functions consider the `dst` array as if it had type:

  ```
  shared <<TYPE>> *
  ```

- The `upc_all_prefix_reduceT_team` function considers the `dst` array as if it had type:

```
shared [blk_size] <<TYPE>> [<nelems>]
```

Get-put-priv Variants

Three variants are defined for each of the three functions presented before. The *get* variant uses a private pointer as destination, the *put* variant uses a private pointer as source, and the *priv* variant uses private pointers as source and destination. Each shared source or destination in these variants is considered to have the same behavior as in the corresponding base collective according to the previous description.

Examples

In the next piece of code, the accumulative sum of all the elements from array `A` associated to threads in team `t` is computed in array `B`, and then the sum of these elements in array `B` is computed in variable `c`.

```
#define NELEMS 1000
shared [THREADS] int A[NELEMS*THREADS];
shared [THREADS] int B[NELEMS*THREADS];
shared int c;
team t;   // The team has to be initialized

// Initialize A
for (int i=MYTHREAD*NELEMS; i<(MYTHREAD+1)*NELEMS; i++) {
    A[i]=i;
}

upc_all_prefix_reduceI_team(B, A, UPC_ADD, NELEMS, NELEMS, NULL, t,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
upc_all_reduceI_team(&c, B, UPC_ADD, NELEMS, NELEMS, NULL, t,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

A.4. Get-put-priv Variants of Standard Collectives

Synopsis

```
#include <upc.h>
#include <upc_collective.h>
/**
 * BROADCAST
 */
void upc_all_broadcast_get (
        void *dst, shared const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_broadcast_put (
        shared void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_broadcast_priv (
        void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
/**
 * SCATTER
 */
void upc_all_scatter_get (
        void *dst, shared const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_scatter_put (
        shared void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_scatter_priv (
        void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
/**
 * GATHER
 */
void upc_all_gather_get (
        void *dst, shared const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_put (
        shared void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
```

```
void upc_all_gather_priv (
        void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
/**
 * ALLGATHER
 */
void upc_all_gather_all_get (
        void *dst, shared const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_all_put (
        shared void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_all_priv (
        void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
/**
 * EXCHANGE
 */
void upc_all_exchange_get (
        void *dst, shared const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_exchange_put (
        shared void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
void upc_all_exchange_priv (
        void *dst, const void *src, size_t nbytes, upc_flag_t sync_mode
);
/**
 * PERMUTE
 */
void upc_all_permute_get (
        void *dst, shared const void *src, shared const int *perm,
        size_t nbytes, upc_flag_t sync_mode
);
void upc_all_permute_put (
        shared void *dst, const void *src, shared const int *perm,
        size_t nbytes, upc_flag_t sync_mode
);
void upc_all_permute_priv (
        void *dst, const void *src, shared const int *perm,
        size_t nbytes, upc_flag_t sync_mode
```

```
);
/**
 * REDUCE
 */
void upc_all_reduce<<T>>_get (
         void *dst, shared const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_put (
         shared void *dst, const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_priv (
         void *dst, const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
         upc_flag_t sync_mode
);
/**
 * ALLREDUCE
 */
void upc_all_reduce<<T>>_all_get (
         void *dst, shared const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_put (
         shared void *dst, const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
         upc_flag_t sync_mode
);
void upc_all_reduce<<T>>_all_priv (
         void *dst, const void *src, upc_op_t op, size_t nelems,
         size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
         upc_flag_t sync_mode
);
/**
 * PREFIX REDUCE
 */
void upc_all_prefix_reduce<<T>>_get (
```

```
        void *dst, shared const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
        upc_flag_t sync_mode
);
void upc_all_prefix_reduce<<T>>_put (
        shared void *dst, const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
        upc_flag_t sync_mode
);
void upc_all_prefix_reduce<<T>>_priv (
        void *dst, const void *src, upc_op_t op, size_t nelems,
        size_t blk_size, <<TYPE>> (*func) (<<TYPE>>,<<TYPE>>),
        upc_flag_t sync_mode
);
```

Description

- These collectives present the same features as the corresponding standard ones, but they use private arrays as source and/or destination of communications.
- The *get-put-priv* variants of the extended collectives have already been included in their corresponding sections: in-place (Section A.1), vector-variant (Section A.2) and team-based collectives (Section A.3).

Bibliography

[1] R. Alameh, N. Zazwork, and J. Hollingsworth. Performance Measurement of Novice HPC Programmers Code. In *Proc. 3rd Intl. Workshop on Software Engineering for High Performance Computing Applications (SE-HPC'07), Minneapolis (MS, USA)*, pages 3–8, 2007. pages 26

[2] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford Science Publications, Oxford, 1987. pages 109

[3] Berkeley Unified Parallel C (UPC) Project. http://upc.lbl.gov [Last visited: January 2013]. pages 13, 22, 28, 73

[4] D. Bonachea. UPC Collectives Value Interface, v1.2. http://upc.lbl.gov/docs/user/README-collectivev.txt, 2005. [Last visited: January 2013]. pages 25

[5] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997. pages 45

[6] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *Proc. 18th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe (NM, USA)*, page 254a, 2004. pages 19

[7] C. Canuto, M. Y. Hussaini, and A. Quarteroni. *Spectral Methods in Fluid Dynamics*. Springer, Berlin, 1988. pages 111

[8] I. Christadler, G. Erbacci, and A. D. Simpson. Performance and Productivity of New Programming Languages. In *Facing the Multicore - Challenge II*, volume 7174 of *Lecture Notes in Computer Science*, pages 24–35. Springer-Verlag, Berlin, 2012. pages 10

[9] Co-Array Fortran. http://www.co-array.org [Last visited: January 2013]. pages 10

[10] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05), Chicago (IL, USA)*, pages 36–47, 2005. pages 19

[11] Cray Inc. Cray Unified Parallel C (UPC). http://docs.cray.com/books/S-2179-50/html-S-2179-50/z1035483822pvl.html [Last visited: January 2013]. pages 14

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI'04), San Francisco (CA, USA)*, pages 137–150, 2004. pages 88

[13] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Communications of the ACM*, 53(1):72–77, 2010. pages 88

[14] L. Dematte. Smoldyn on Graphics Processing Units: Massively Parallel Brownian Dynamics Simulations. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3):655–667, 2012. pages 107

[15] Development Time Working Group - HPCS. UMDInst. http://code.google.com/p/umdinst/ [Last visited: January 2013]. pages 26

[16] J. Dinan, P. Balaji, E. L. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proc. 7th ACM Intl. Conference on Computing Frontiers (CF'10), Bertinoro (Italy)*, pages 177–186, 2010. pages 53

[17] M. Długosz, P. Zieliński, and J. Trylska. Brownian Dynamics Simulations on CPU and GPU with BD_BOX. *Journal of Computational Chemistry*, 32(12):2734–2744, 2011. pages 107

[18] H. Dong, S. Zhou, and D. Grove. X10-Enabled MapReduce. In *Proc. 4th Conference on Partitioned Global Address Space Programming Models (PGAS'10), New York (NY, USA)*, pages 9:1–9:6. ACM, 2010. pages 88

[19] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: a NPB Experimental Study. In *Proc. 15th ACM/IEEE Conference for High Performance Computing, Networking, Storage and Analysis (SC'02), Baltimore (MD, USA)*, pages 1–26, 2002. pages 19

[20] T. 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):764–775, 2006. pages 19, 69

[21] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2005. pages 9

[22] T. El-Ghazawi and S. Chauvin. UPC Benchmarking Issues. In *Proc. 30th Intl. Conference on Parallel Processing (ICPP'01), Valencia (Spain)*, pages 365–372, 2001. pages 19, 60, 77, 89, 115

[23] D. L. Ermak and J. A. McCammon. Brownian Dynamics with Hydrodynamic Interactions. *Journal of Chemical Physics*, 69(4):1352–1360, 1978. pages 107

[24] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta. Measuring HPC Productivity. *Intl. Journal of High Performance Computing Applications*, 18(4):459–473, 2004. pages 26

[25] Finis Terrae Supercomputer at CESGA. https://www.cesga.es/en/infraestructuras/computacion/finisterrae [Last visited: January 2013]. pages 23

[26] M. Fixman. Construction of Langevin Forces in the Simulation of Hydrodynamic Interaction. *Macromolecules*, 19(4):1204–1207, 1986. pages 109

[27] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962. pages 28

[28] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Application of a Development Time Productivity Metric to Parallel Software Development. In *Proc. 2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS'05), St. Louis (MO, USA)*, pages 8–12, 2005. pages 26

[29] Galicia Supercomputing Center (CESGA). http://www.cesga.es/en/inicio [Last visited: January 2013]. pages 26

[30] J. García de la Torre, J. G. Hernández Cifre, A. Ortega, R. R. Schmidt, M. X. Fernandes, H. E. Pérez Sánchez, and R. Pamies. SIMUFLEX: Algorithms and Tools for Simulation of the Conformation and Dynamics of Flexible Molecules and Nanoparticles in Dilute Solution. *Journal of Chemical Theory and Computation*, 5(10):2606–2618, 2009. pages 107

[31] J. Gustafson. Purpose-Based Benchmarks. *Intl. Journal of High Performance Computing Applications*, 18(4):475–487, 2004. pages 18

[32] GWU Unified Testing Suite (GUTS). http://threads.hpcl.gwu.edu/sites/guts [Last visited: January 2013]. pages 69, 77

[33] Hadoop Project. http://hadoop.apache.org [Last visited: January 2013]. pages 88

[34] Hewlett-Packard Inc. HP Unified Parallel C (HP UPC). http://hp.com/go/upc/ [Last visited: January 2013]. pages 14

[35] High Productivity Computer Systems. http://www.highproductivity.org [Last visited: January 2013]. pages 10

[36] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards Efficient MapReduce Using MPI. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo (Finland)*, pages 240–249, 2009. pages 88

[37] G. A. Huber and J. A. McCammon. Browndye: A Software Package for Brownian Dynamics. *Computer Physics Communications*, 181(11):1896–1905, 2010. pages 107

[38] InfiniBand Trade Association. http://www.infinibandta.org [Last visited: January 2013]. pages 23

[39] Intrepid Technology Inc. GNU Unified Parallel C (GNU UPC). http://www.gccupc.org [Last visited: January 2013]. pages 13

[40] R. M. Jendrejack, M. D. Graham, and J. J. de Pablo. Hydrodynamic Interactions in Long Chain Polymers: Application of the Chebyshev Polynomial Approximation in Stochastic Simulations. *Journal of Chemical Physics*, 113(7):2894–2900, 2000. pages 111

[41] J. Kepner. High Performance Computing Productivity Model Synthesis. *Intl. Journal of High Performance Computing Applications*, 18(4):505–516, 2004. pages 18

[42] P. Krajca and V. Vychodil. Distributed Algorithm for Computing Formal Concepts Using Map-Reduce Framework. In *Proc. 8th Intl. Symposium on Intelligent Data Analysis (IDA'09), Lyon (France)*, pages 333–344, 2009. pages 87

[43] E. Lusk and K. Yelick. Languages for High-Productivity Computing: the DARPA HPCS Language Project. *Parallel Processing Letters*, 17(1):89–102, 2007. pages 10, 18

[44] 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. 3rd Conference on Partitioned Global Address Space Programming Models (PGAS'09), Ashburn (VA, USA)*, pages 9:1–9:7. ACM, 2009. pages xviii, 142

[45] 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), Hamburg (Germany)*, 2010 (Poster). pages 73

[46] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño. Performance Evaluation of

MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo (Finland)*, pages 174–184, 2009. pages xviii, 69, 142

[47] J. Manzano, Y. Zhang, and G. Gao. P3I: The Delaware Programmability, Productivity and Proficiency Inquiry. In *Proc. 2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS'05), St. Louis (MO, USA)*, pages 32–36, 2005. pages 26

[48] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for Multicore Architectures. In *Technical Report MIT-CSAIL-TR-2010-020, MIT, USA*, 2010. pages 88

[49] MapReduce-MPI Library. http://mapreduce.sandia.gov [Last visited: January 2013]. pages 94

[50] S. J. Matthews and T. L. Williams. MrsRF: an Efficient MapReduce Algorithm for Analyzing Large Collections of Evolutionary Trees. *BMC Bioinformatics*, 11(Suppl. 1):S15, 2010. pages 87

[51] R. M. C. McCreadie, C. Macdonald, and I. Ounis. On Single-Pass Indexing with MapReduce. In *Proc. 32nd Intl. ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'09), Boston (MA, USA)*, pages 742–743, 2009. pages 87

[52] Message-Passing Interface Forum. http://www.mpi-forum.org [Last visited: January 2013]. pages 18

[53] Michigan Technological University. MuPC: A Run Time System for Unified Parallel C. http://www.upc.mtu.edu/mupc.html [Last visited: January 2013]. pages 14

[54] Michigan Technological University. UPC Collectives Reference Implementation. http://www.upc.mtu.edu/collectives/col1.html [Last visited: January 2013]. pages 22

[55] R. Murty and D. Okunbor. Efficient Parallel Algorithms for Molecular Dynamics Simulations. *Parallel Computing*, 25(3):217–230, 1999. pages 118

[56] NASA Advanced Computing Division. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/ [Last visited: January 2013]. pages 69

[57] R. Nishtala, G. Almási, and C. Caşcaval. Performance without Pain = Productivity: Data Layout and Collective Communication in UPC. In *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08), Salt Lake City (UT, USA)*, pages 99–110, 2008. pages 19, 53

[58] R. Nishtala, Y. Zheng, P. Hargrove, and K. Yelick. Tuning Collective Communication for Partitioned Global Address Space Programming Models. *Parallel Computing*, 37(9):576–591, 2011. pages 25

[59] NIST Matrix Market. FIDAP011: Matrices Generated by the FIDAP Package. http://math.nist.gov/MatrixMarket/data/SPARSKIT/fidap/fidap011.html [Last visited: January 2013]. pages 79

[60] Northrop Grumman Corporation - IT Solutions. CLOC - Count Lines Of Code. http://cloc.sourceforge.net [Last visited: January 2013]. pages 28

[61] ParTec. ParaStation MPI. http://www.par-tec.com/products/parastation-mpi.html [Last visited: January 2013]. pages 84, 96, 126

[62] I. Patel and J. R. Gilbert. An Empirical Study of the Performance and Productivity of Two Parallel Programming Models. In *Proc. 22nd IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'08), Miami (FL, USA)*, pages 1–7, 2008. pages 26

[63] S. J. Plimpton and K. D. Devine. MapReduce in MPI for Large-scale Graph Algorithms. *Parallel Computing*, 37(9):610–632, 2011. pages 88

[64] Pluton Cluster - Department of Electronics and Systems, University of A Coruña. http://pluton.des.udc.es [Last visited: January 2013]. pages 28

[65] Project Fortress. http://projectfortress.java.net [Last visited: January 2013]. pages 11

[66] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. 13th Intl. Conference on High-Performance Computer Architecture (HPCA'07), Phoenix (AZ, USA)*, pages 13–24, 2007. pages 88, 94

[67] J. Rotne and S. Prager. Variational Treatment of Hydrodynamic Interaction in Polymers. *Journal of Chemical Physics*, 50(11):4831–4837, 1969. pages 109

[68] Z. Ryne and S. Seidel. A Specification of the Extensions to the Collective Operations of Unified Parallel C. http://www.upc.mtu.edu/papers/cstr05-08.pdf [Last visited: January 2013]. pages 24, 25

[69] 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: January 2013]. pages 25

[70] Z. Ryne and S. Seidel. UPC Extended Collective Operations Specification. http://www.upc.mtu.edu/papers/UPC_CollExt.pdf [Last visited: January 2013]. pages 24, 53

[71] R. A. Salama and A. Sameh. Potential Performance Improvement of Collective Operations in UPC. *Advances in Parallel Computing*, 15:413–422, 2008. pages 25

[72] A. Satoh. *Introduction to Molecular-Microsimulation for Colloidal Dispersions*. Elsevier, 2003. pages 109

[73] T. Schlick, D. A. Beard, J. Huang, D. Strahs, and X. Qian. Computational Challenges in Simulating Large DNA over Long Times. *IEEE Computing in Science and Engineering*, 2(6):38–51, 2000. pages 111

[74] M. Snir and D. Bader. A Framework for Measuring Supercomputer Productivity. *Intl. Journal of High Performance Computing Applications*, 18(4):417–432, 2004. pages 18

[75] SPAM Corpus - Text Retrieval Conference (TREC). http://plg.uwaterloo.ca/~gvcormac/treccorpus/ [Last visited: January 2013]. pages 95

[76] SVG supercomputer at CESGA. https://www.cesga.es/en/infraestructuras/computacion/svg [Last visited: January 2013]. pages 125

[77] 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. 11th IEEE Intl. Conference on High Performance Computing and Communications (HPCC'09), Seoul (Korea)*, pages 69–78, 2009. pages xviii, 21, 142

[78] C. Teijeiro, G. Sutmann, G. L. Taboada, and J. Touriño. Parallelization and Performance Analysis of a Brownian Dynamics Simulation using OpenMP. In *Proc. 12th Intl. Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'12), La Manga del Mar Menor (Spain)*, pages 1143–1154, 2012. pages xviii, 142

[79] 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*, 2013 (in press). pages xviii, 142

[80] 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. *Journal of Supercomputing*, 2013 (in press). pages xviii, 142

[81] C. Teijeiro, G. L. Taboada, J. Touriño, and R. Doallo. Design and Implementation of MapReduce using the PGAS Programming Model with UPC. In *Proc. 17th Intl. Conference on Parallel and Distributed Systems (ICPADS'11), Tainan (Taiwan)*, pages 177–186, 2011. pages xviii, 87, 142

[82] C. Teijeiro, G. L. Taboada, J. Touriño, R. Doallo, J. C. Mouriño, D. A. 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. pages xviii, 41, 142

[83] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC Programmability using Classroom Studies. In *Proc. 3rd Conference on Partitioned Global Address Space Programming Models (PGAS'09), Ashburn (VA, USA)*, pages 10:1–10:7. ACM, 2009. pages xviii, 26, 142

[84] The Chapel Parallel Programming Language. http://chapel.cray.com [Last visited: January 2013]. pages 10

[85] The OpenMP API Specification for Parallel Programming. http://www.openmp.org [Last visited: January 2013]. pages 18

[86] The Phoenix System for MapReduce Programming. http://mapreduce.stanford.edu [Last visited: January 2013]. pages 94

[87] Titanium Project Home Page. http://titanium.cs.berkeley.edu [Last visited: January 2013]. pages 10

[88] UC Berkeley. GASNet Communication System. http://gasnet.cs.berkeley.edu [Last visited: January 2013]. pages 13, 22

[89] Unified Parallel C (UPC) Programming Language Specification Development. http://code.google.com/p/upc-specification/ [Last visited: January 2013]. pages 38

[90] Universidade da Coruña (UDC). http://www.udc.es [Last visited: January 2013]. pages 26

[91] UPC Activities at the University of Florida. http://www.hcs.ufl.edu/upc/ [Last visited: January 2013]. pages 11

[92] UPC Consortium. UPC Collective Operations Specifications v1.0. http://upc.gwu.edu/docs/UPC_Coll_Spec_V1.0.pdf [Last visited: January 2013]. pages 21

[93] UPC Consortium. UPC Language Specifications v1.2. http://upc.gwu.edu/docs/upc_specs_1.2.pdf [Last visited: January 2013]. pages 21, 143

[94] UPC Operations Microbenchmarking Suite. http://forge.cesga.es/projects/uoms [Last visited: January 2013]. pages 73

[95] UPC Projects at MTU. http://www.upc.mtu.edu [Last visited: January 2013]. pages 11

[96] UPC Wiki - Main Page. http://upc.lbl.gov/wiki [Last visited: January 2013]. pages 11

[97] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proc. 19th Intl. Symposium on Computer Architecture (ISCA'92), Gold Coast (Australia)*, pages 256–266, 1992. pages 22

[98] Webb Spam Corpus. http://www.cc.gatech.edu/projects/doi/WebbSpam Corpus.html [Last visited: January 2013]. pages 95

[99] X10: Performance and Productivity at Scale. http://x10-lang.org [Last visited: January 2013]. pages 10

[100] K. Yelick. Lecture Notes on Global Address Space Programming in UPC - Applications of Parallel Computing (CS267) at UC Berkeley. http://crd.lbl.gov/~dhbailey/cs267/lecture13_upc_ky08.pdf [Last visited: January 2013]. pages 27

[101] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proc. 1st Intl. Workshop on Parallel Symbolic Computation (PASCO'07), London (ON, Canada)*, pages 24–32, 2007. pages 19

[102] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *Proc. 2009 IEEE Intl. Symposium on Workload Characterization (IISWC'09), Austin (TX, USA)*, pages 198–207, 2009. pages 88

[103] Y. Zhang, J. J. de Pablo, and M. D. Graham. An Immersed Boundary Method for Brownian Dynamics Simulation of Polymers in Complex Geometries: Application to DNA Flowing through a Nanoslit with Embedded Nanopits. *Journal of Chemical Physics*, 136(1):014901, 2012. pages 106

[104] Z. Zhang and S. Seidel. Benchmark Measurements of Current UPC Platforms. In *Proc. 19th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'05), Denver (CO, USA)*, 2005. pages 19, 69