Universidad de Valladolid

Escuela Técnica Superior de
Ingeniería Informática

Departamento de Informática

Tesis Doctoral:

Easing Parallel Programming on Heterogeneous Systems

Presentada por Ana Moretón Fernández para optar al grado
de doctor por la Universidad de Valladolid

Dirigida por:
Dr. Arturo González Escribano

Valladolid, 2018

Resumen

El modo más frecuente de resolver aplicaciones de HPC (High performance Computing) en tiempos de ejecución razonables y de una forma escalable es mediante el uso de sistemas de cómputo paralelo. La tendencia actual en los sistemas de HPC es la inclusión en la misma máquina de ejecución de varios dispositivos de cómputo, de diferente tipo y arquitectura. Sin embargo, su uso impone al programador retos específicos. Un programador debe ser experto en las herramientas y abstracciones existentes para memoria distribuida, los modelos de programación para sistemas de memoria compartida, y los modelos de programación específicos para para cada tipo de co-procesador, con el fin de crear programas híbridos que puedan explotar eficientemente todas las capacidades de la máquina. Actualmente, todos estos problemas deben ser resueltos por el programador, haciendo así la programación de una máquina heterogénea un auténtico reto.

Esta Tesis trata varios de los problemas principales relacionados con la programación en paralelo de los sistemas altamente heterogéneos y distribuidos. En ella se realizan propuestas que resuelven problemas que van desde la creación de códigos portables entre diferentes tipos de dispositivos, aceleradores, y arquitecturas, consiguiendo a su vez máxima eficiencia, hasta los problemas que aparecen en los sistemas de memoria distribuida relacionados con las comunicaciones y la partición de estructuras de datos.

Palabras clave

Computación paralela, Entornos paralelos, Sistemas heterogeneos, Nuevos modelos de programación, Memoria distribuida, Cálculo de comunicaciones

Abstract

The use of parallel computing systems frequently represents the only scalable way to solve HPC (High performance Computing) problems in reasonable execution times. The current trend in high performance computing platforms is to include in the same machine several parallel devices, of different type and architectures, and to interconnect them to form highly parallel and heterogeneous distributed systems. Programming efficient and portable parallel applications that can really exploit these systems, imposes specific and complex challenges to the programmers. A programmer must be proficient in distributed-memory communication tools or layers, shared-memory programming models, and specific programming models for the available co-processors, in order to create hybrid programs that will exploit all the machine capabilities. Moreover, she also has to deal with the proper workload distribution among the different nodes and devices, assigning to each one an amount of workload related to their computation power and features. Nowadays, all these issues should be solved by the programmer, making the programming of heterogeneous platforms an actual challenge.

This PhD. Thesis addresses several main problems related to the parallel programming for highly heterogeneous and distributed systems. It first tackles problems to allow the developing of efficient coordination codes, portable across different kind of devices, accelerators, and architectures. Then, it also targets problems related to the data communication and partition issues concerning the use of devices in distributed-memory systems. In this dissertation we introduce abstractions, mechanisms, and methods to solve many of these problems. We also discuss their practical application to develop research prototypes and actual programming tools. Experimental works conducted using these tools validates the applicability of the proposed techniques and the portability, efficiency, and versatility of the programs that can be obtained.

Keywords

Parallel programming, Parallel frameworks, Heterogeneous systems, New programming models, Distributed-memory, Communication calculation.

This research has been partially supported by Universidad de Valladolid (UVa), MICINN (Spain), the ERDF program of the European Union and Junta de Castilla y Leon: HomProg-HetSys project (TIN2014-58876-P), PCAS (TIN 2017-88614-R), CAPAP-H5 network (TIN2014-53522-REDT), CAPAP-H6 (TIN2016-81840-REDT), COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS), FEDER Grant VA082P17 (PROPHET Project), and by the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain.

Agradecimientos

Tras casi 7 años con ellos (y por aguantarme tanto tiempo yo creo que se lo merecen), empiezo mis agradecimientos a mis tutores Diego y Arturo. En especial dar las gracias a Arturo, por las innumerables horas gastadas juntos escribiendo papers con fórmulas matemáticas inentendibles, por la paciencia que tuvo hasta que aprendí a escribir dos frases seguidas, y por no desistir ni dejar de confiar a pesar de los dos millones de papers rechazados (más o menos echando las cuentas a ojo). Muchas gracias a ambos por darme la oportunidad de realizar el doctorado con vosotros y compartir laboratorio y puesto de trabajo con un motón de gente maravillosa. Gente a la que le debo la Tesis. No se si hubiese sido capaz de acabarla sin los coffe-breaks y la compañía de: Sergio, Alvaro, Héctor, Javi, Yuri, Dani, Edu y un montón más de personas haciendo proyectos fin de carrera, que aunque no pueda nombrar a todas les sigo teniendo muy en cuenta. Tampoco me quiero olvidar de mis compañeros de carrera, con los que empecé este camino, y con los que se me metió esta idea de doctorado en la cabeza. Agradecer también a Ana Lucia Varbanescu la opción de realizar mi estancia de tres meses en Delft, y agradecerla a ella y al resto de compañeros de allí también el trato recibido durante la estancia.

Y cómo no agradecer a mis amigos, empezando por la gente con la que he crecido, con la que mantengo decenas de años de amistad y que en todo momento me han apoyado; y terminando por los Muy Mejores Amig@s y la peña Despiporre que, a pesar de la diversidad de ideas y vidas que hay en el grupo, han sido y son una fuente inagotable de risas, fiestas, alegrías, positivismo y el mejor ejemplo que conozco a seguir como grupo. Como solemos decir: *Poco nos vemos, para lo mucho que nos queremos*.

Por último agradecer a mi familia. Empezaré por orden inverso de antigüedad =). A Adri por estar siempre ahí apoyándome en lo bueno y en lo malo, animarme en este camino siempre que lo he necesitado, y disfrutar conmigo los buenos momentos. A mi hermano, por ser siempre la cabeza pensante, objetiva y razonable de la familia. Seguiría dudando de cualquier tontería de hace 5 años si no fuese por él. A mi madre. Sería banal decir que ella me ha ayudado en este camino, porque en realidad ella me ha ayudado en todos (una Santa como dicen mis amigos). No sería la persona que soy sin ella. Y por último a mi padre, la persona que más orgullosa ha estado de mi, que más me ha apoyado durante los estudios y la que, por qué no decirlo también, la que más presumía de hija. Siempre guardaré tus consejos, y haré todo lo posible para que sigas orgulloso de mi allá donde estés.

Ana Moreton-Fernandez

Contents

List of Figures

List of Tables

Resumen de la Tesis Doctoral

El modo más frecuente de resolver aplicaciones de HPC (High performance Computing) en tiempos de ejecución razonables y de una forma escalable es mediante el uso de sistemas de cómputo paralelo. Sin embargo, su uso enfrenta al programador retos específicos.

Este capítulo describirá esos retos, y de acuerdo a los problemas que se vayan identificando, presentará los objetivos, la pregunta de investigación de esta Tesis Doctoral, y se describirán de forma general las contribuciones aportadas a lo largo de la Tesis Doctoral, para resolver las cuestiones planteadas por la pregunta de investigación, y conseguir los objetivos descritos.

R.1 Motivación

En esta sección presentamos la motivación que da lugar al planteamiento de esta Tesis Doctoral. En primer lugar repasaremos la evolución de los sistemas de computo paralelos. Veremos como esta evolución ha llevado a la necesidad de diseñar nuevos modelos de programación y nuevos sistemas de control y gestión del paralelismo durante la ejecución. Analizaremos diferentes clases de propuestas para dichos modelos y herramientas, y descubriremos cuestiones aún no resueltas. Esta observación nos llevará a plantear los diversos retos que esta Tesis Doctoral trata de resolver. En esta sección introduciremos también el trabajo realizado en este ámbito por el grupo de investigación Trasgo, en el que se ha desarrollado el trabajo principal que se describe en este volumen.

R.1.1 Computación paralela

Existen muchos problemas de cómputo, como por ejemplo los relacionados con las simulaciones médicas, el análisis de tráfico o flujo en redes, o la navegación de coches autónomos, que requieren simulaciones muy complejas, niveles de precisión muy altos o respuestas al problema en un tiempo limitado y generalmente muy corto [9]. Estos requisitos se corresponden con programas de un alto coste computacional y hacen que la ejecución de estos programas sea prohibitiva con programas o máquinas que ejecutan las instrucciones de forma únicamente secuencial.

Una solución para reducir el tiempo de ejecución de este tipo de aplicaciones es el aumento de la frecuencia de reloj en los sistemas hardware que ejecutan los programas. Sin embargo, la evolución de la velocidad de reloj de los procesadores y chips basados en una alta escala de integración de transistores se está viendo limitada. El consumo de energía y el calor disipado aumenta demasiado rápido con la frecuencia con las tecnologías actuales [47]. Por ello, la tendencia actual pasa por utilizar la enorme cantidad de transistores que se puede integrar en un reducido espacio para construir chips con varios elementos de computo que pueden trabajar de forma simultánea o paralela. Así mismo, dentro de la misma máquina se pueden colocar varios dispositivos paralelos, e interconectar con tecnología de red varias máquinas para formar enormes y complejos sistemas capaces de ejecutar multitud de subprogramas o tareas simultáneamente. Por tanto, la división de los trabajos en varias tareas independientes puede ser una solución para los problemas de alto coste computacional mencionados [3]. Cuántas más unidades de procesamiento tenga una máquina, más tareas pueden ser creadas y ejecutadas simultáneamente, reduciendo el tiempo total de ejecución. Este método de diseño de programas nos proporciona soluciones escalables. Así, los problemas que necesitan cada vez más cantidad de cómputo, cuya carga computacional crece, se pueden resolver en tiempos razonables con un número proporcionalmente mayor de elementos de cómputo. Esta característica ha hecho que hoy en día los sistemas paralelos se estén convirtiendo en la base de la computación de alto coste o alto rendimiento (HPC, High-Performance Computing).

La necesidad de dar con una solución para los problemas de HPC ha fomentado el desarrollo de los sistemas paralelos, haciendo que durante las últimas décadas el número de elementos de procesamiento en los sistemas paralelos vaya aumentando. Por otra parte, el tiempo y el esfuerzo necesario para el desarrollo de programas para su ejecución en paralelo ha crecido desmesuradamente [56] debido a: (1) la inherente complejidad de crear y coordinar las tareas a ejecutar por cada elemento de procesamiento; (2) la falta de modelos de programación, herramientas de desarrollo y estrategias de diseño que realmente permitan la portabilidad de los códigos entre sistemas diferentes; y (3) la gran diversidad de tipos y arquitecturas en las plataformas de cómputo paralelo.

A lo largo de esta sección repasaremos las arquitecturas de las máquinas más usadas actualmente para la computación en paralelo. Después, describiremos los modelos de programación más conocidos y usados para cada arquitectura.

R.1.2 Sistemas para la computación paralela

La evolución de los sistemas de cómputo se ha focalizado en el diseño de nuevas arquitecturas con un creciente nivel de paralelismo, en lugar de en el desarrollo de procesadores más rápidos. La computación paralela representa en muchas situaciones la única solución escalable para resolver ciertos problemas en tiempos de ejecución razonables.

Existen diversos términos y denominaciones referidos a sistemas cuyas arquitecturas incluyen múltiples elementos de cómputo independientes. El término *multi-core* se refiere a la integración de varias unidades de procesamiento en un único chip, donde además cada unidad de procesamiento puede operar como un procesador de propósito general. El uso de varias unidades de procesamiento a la vez implica un aumento del rendimiento, sin la necesidad de aumentar la frecuencia de reloj, debido a que la carga de trabajo puede ser distribuida entre los diversos elementos de procesamiento. De acuerdo a su diseño, los elementos de procesamiento pueden compartir espacio de memoria, formando una memoria global, o no. La presencia de un único espacio de memoria global hace la programación de estas plataformas más natural para los programadores habituados a los modelos clásicos de programación secuencial. Las operaciones de memoria de solo lectura serán invisibles para el programador, y programadas de forma similar que en los programas secuenciales. Sin embargo, las operaciones de escritura requieren que el programador use sistemas de exclusión mutua para los accesos concurrentes, o sincronizaciones de lectura-escritura, haciendo su tarea mucho más complicada.

Por otra parte, los sistemas que no comparten espacio de memoria presentan retos aparentemente diferentes al programador. Cada unidad de procesamiento de una máquina puede tener su propio espacio de memoria local. Así, todas las interacciones entre diferentes elementos de procesamiento deben ser realizadas, por ejemplo, mediante el uso de abstracciones de comunicación como son los mensajes, los cuales podrán transferir datos, trabajo o simplemente sincronizar acciones entre los procesos.

Coprocesadores para la computación paralela

Un coprocesador puede ser considerado como un dispositivo de computación que se añade a un sistema por otra parte completo, con el fin de ejecutar software especializado. Los coprocesadores más usados para supercomputación hoy en día son las GPUs (Graphics Processing Unit) [133]. Las GPUs fueron originalmente diseñadas para ayudar al procesador principal en el procesamiento gráfico. En las aplicaciones de procesamiento gráfico, la reutilización de datos es pequeña, y los programas son muy simples. Por ello, el diseño de las GPUs está basado en el uso de pequeñas cachés on-chip con una colección de Unidades Aritmético-Lógicas (ALUs) sencillas. El uso de coprocesadores como procesadores de propósito general ha sido tendencia en la supercomputación desde que Nvidia lanzase la tarjeta Tesla en 2007. Aunque las GPUs [133] fueron originalmente diseñadas para el procesamiento gráfico, sus potentes capacidades de cómputo masivamente paralelo han hecho su uso muy popular en las aplicaciones de HPC, creando una nueva tendencia basada en ellas, GPGPU (General-Purpose Computing on Graphics Processing Units).

Durante los últimos años han surgido nuevos dispositivos aceleradores para atacar eficientemente el tipo de problemas que no encajan bien con la arquitectura de las GPUs. Un ejemplo es la familia de tarjetas Intel Xeon Phi (lanzada en 2012), la cual también es conocida como Many Integrated Cores (MICs) [73]. El coprocesador Xeon Phi es un acelerador con muchos cores de propósito general basados en la tecnología x86, con unidades de vectorización mejoradas y mejor ancho de banda de memoria.

Computación Heterogénea: Sacando partido a todos los tipos de arquitecturas a la vez.

La evolución de los supercomputadores puede seguirse gracias al proyecto Top500 [130], que cada seis meses publica una clasificación de los 500 computadores más potentes en el mundo, en base a su velocidad de ejecución de unos programas de prueba concretos. Los supercomputadores actuales están compuestos por varios nodos (máquinas interconectadas) con memoria distribuida donde cada nodo puede tener diferentes capacidades de cómputo, jerarquías de memoria o coprocesadores asociados como GPUs o Xeon Phis. En muchos casos, las características de computación de cada nodo son muy diferentes. Este tipo de sistemas son denominados *Sistemas Heterogéneos* [26].

R.1.3 Modelos de programación paralelos

Según el trabajo de Balaji [12], un modelo de programación puede entenderse como una máquina abstracta para que un programador escriba instrucciones. Típicamente, estos modelos de programación son instanciados en lenguajes en si mismos, o en bibliotecas de funciones.

Esta nueva era de la computación en paralelo está requiriendo un continuo esfuerzo de investigación en el diseño y desarrollo de nuevos modelos de programación, apropiados para explotar la computación paralela en aplicaciones de alto coste computacional. Programar

sistemas paralelos es complicado. Con el fin de lograr una ejecución correcta del programa y que realmente aproveche el nivel de paralelismo que ofrecen las arquitecturas modernas, el programador debe razonar teniendo en cuenta los comportamientos estocásticos asociados con la ejecución simultánea y los costes asociados a los movimientos de datos entre múltiples unidades de procesamiento y entre sus diferentes niveles en la jerarquía de memoria. Así, los modelos de programación paralelos definen de forma explícita o implícita dos submodelos. El primero es un modelo de ordenación y sincronización entre las secuencias de operaciones que el código ejecutará, generalmente llamado *modelo de ejecución*. El segundo es un modelo que determina como mover los datos entre los nodos y las jerarquías de memoria del sistema paralelo, llamado *modelo de memoria*.

Existen cuatro pilares o características fundamentales que representan las posibles preferencias del programador entre las características de un modelo de programación, independientemente de que sea secuencial o paralelo: (1) *Productivo*, capaz de expresas algoritmos abstractos con facilidad, (2) *Portable*, capaz de ser usado en cualquier arquitectura de computación, (3) *Eficiente*, capaz de producir un buen rendimiento adecuado con el hardware donde se esta ejecutando, y (4) *Expresivo*, capaz de representar o expresar un gran rango de clases de algoritmos.

Sería ideal diseñar un lenguaje de programación que cumpla con las cuatro características expuestas, sin embargo esto parece imposible por el momento. Por ello, existen muchos lenguajes de programación secuencial, cada uno aportando diferentes soluciones a la hora de expresar algoritmos y transformarlos a código ejecutable, focalizándose en una combinación particular de las anteriores características. Este comportamiento no es diferente para los sistemas paralelos. Los diferentes usuarios tienen preferencias entre los niveles de abstracción que nos dan las combinaciones de los cuatro pilares. En esta sección veremos un resumen de los modelos de programación paralela más conocidos y usados para HPC y la supercomputación actualmente.

Clasificación de los modelos de programación paralela

Muchos modelos y lenguajes de programación han surgido para la programación en paralelo, con el fin de abstraer al programador de muchos de los problemas de implementación que típicamente aparecen en este tipo de computo. Algunos nos abstraen por completo del modelo de arquitectura, pero suelen tener problemas de eficiencia en la implementación sobre arquitecturas completas. Para poder utilizar formas de implementación más eficientes y específicas, otros se focalizan en alguno de los modelos fundamentales de arquitectura, como son por ejemplo los modelos de memoria compartida, o de memoria distribuida. OpenMP y el paradigma de paso de mensajes (implementado por ejemplo en bibliotecas MPI) son los métodos de programación paralela más extendidos y usados para memoria compartida y distribuida respectivamente [32, 59]. Ambas aproximaciones han sido declaradas como los modelos más básicos de programación paralela, y dominan el campo de la programación para HPC desde finales de los años 90 del siglo pasado [44].

OpenMP [33, 40] es una API (application programming interface) que facilita la programación en paralelo para sistemas de memoria compartida. Está compuesta por un conjunto de directivas, pragmas, funciones de biblioteca y un sistema de tiempo de ejecución que es capaz de manejar la creación de hilos, su sincronización y por tanto de sus operaciones, y las interacciones que se producen en las operaciones en memoria compartida. Originalmente, OpenMP fue diseñado para paralelizar códigos secuenciales con un bajo esfuerzo de programación. Los códigos secuenciales son anotados con una serie de pragmas o anotaciones que no son parte del lenguaje, pero incluyen información extra para el compilador. Estas anotaciones son procesadas por el compilador, que debe tener la tecnología adecuada para entenderlas y utilizarlas para producir un código que resuelva los problemas de la paralelización siguiendo las pautas indicadas en los pragmas. Originalmente OpenMP estaba orientado a la paralelización de iteraciones de bucles. Actualmente, OpenMP también provee herramientas orientadas a tareas dinámicas y estructuras para la programación paralela más complejas [10].

Existen también otros modelos menos abstractos para memoria compartida como POSIX threads (PThreads) [29, 78], que es la herramienta en la que se apoyan muchos compiladores de OpenMP para la gestión de las tareas a bajo nivel. Se trata de un estándar que define un conjunto de tipos y funciones, para el que se han implementado bindings en los lenguajes de programación más populares, como por ejemplo C/C++, Fortran, Python, etc. La biblioteca de Pthreads nos da una serie de funciones para la creación y destrucción de hilos y para la coordinación de las actividades de los hilos. Sin embargo, comparado con OpenMP, la interfaz de Pthreads es de un nivel más bajo, necesitando el programador un mayor esfuerzo de desarrollo.

La programación en sistemas de memoria distribuida impone retos específicos. El *paso de mensajes* es el modelo de programación más extendido para estos sistemas. Toda la información necesaria para la ejecución del proceso local, que pueda estar contenida en un proceso remoto, debe ser enviada antes al proceso local mediante un intercambio de mensajes. El estándar MPI ha dominado el campo de las propuestas de este modelo, existiendo implementaciones muy eficientes, y siendo hoy en día el estándar de facto para las aplicaciones de HPC en arquitecturas distribuidas [58, 59, 89]. Sin embargo, cuando el programador usa las especificaciones de MPI aun tiene que tratar con problemas de diseño e implementación complejos, como son las decisiones acerca de la partición y la localidad de datos vs. la sincronización y los costes de las comunicaciones. No solo MPI trata con los retos de la programación en sistemas distribuidos. También existen soluciones como GASnet [20] que proveen una capa de programación de bajo nivel para la implementación de lenguajes y bibliotecas paralelas con un espacio de memoria global (modelos PGAS), que proveen de una capa de abstracción similar a la memoria compartida, con el objetivo de simplificar la programación a los usuarios finales.

La programación de co-procesadores es diferente de la programación de sistemas multicore o en memoria distribuida en muchos aspectos. El programador tiene que tratar con muchos detalles manualmente, como por ejemplo las transferencias de datos al dispositivo, o

la configuración de los parámetros de ejecución de cada bloque paralelo. NVIDIA presentó en 2007 CUDA [102], un nuevo modelo de programación especialmente diseñado para los aceleradores GPU de NVIDIA. El modelo de programación CUDA solo se puede utilizar actualmente con los aceleradores GPU de NVIDIA. Para otros tipos de GPUs existen otros modelos de programación similares como son OpenCL [127] o BrookGPU [27]. Para otros tipos de aceleradores como la Xeon Phi, modelos de programación basados en pragmas, como LEO [101], ofrecen mecanismos de programación similares.

Programación paralela para sistemas heterogéneos multinivel.

Diferentes modelos de programación han sido establecidos para cada diferente tipo de dispositivo o arquitectura como hemos descrito previamente. Esto nos indica, que para aprovechar los sistemas paralelos heterogéneos actuales, un programador debe ser experto en las herramientas y abstracciones existentes para memoria distribuida (e.g MPI, GASNet [20], etc), los modelos de programación para sistemas de memoria compartida (e.g OpenMP), y los modelos de programación específicos para para cada tipo de co-procesador (e.g CUDA, OpenCL, o tecnologías equivalentes), con el fin de crear programas híbridos que puedan explotar eficientemente todas las capacidades de la máquina [141]. Además, el programador también tiene que tratar con la distribución de trabajo apropiada para cada caso entre los diferentes nodos y dispositivos, asignando a cada uno, una carga de trabajo proporcional a sus capacidades y capacidad de cómputo. Actualmente, todos estos problemas deben ser resueltos por el programador, haciendo así la programación de una máquina heterogénea un auténtico reto.

R.2 Objetivos de la Tesis Doctoral

De acuerdo con los problemas encontrados y descritos en la sección anterior, la pregunta de investigación que esta Tesis doctoral trata de resolver es la siguiente:

> *Es posible desarrollar: (1) Abstracciones simples que hagan transparente y uniforme el desarrollo, despliegue y coordinación de programas entre diferentes sistemas y arquitecturas paralelas de cómputo, y (2) nuevas técnicas de tiempo de ejecución que, basándose en las dependencias de datos de un programa, puedan distribuir y comunicar de forma automática los datos, de forma transparente para el programador, una vez que las características de la plataforma de ejecución sean conocidas?*

R.2.1 Metodología de investigación

La metodología de investigación usada en esta Tesis esta basada en el método experimental y el método de ingeniería software, compuestos ambos for cuatro fases: Observar la

solución existente, proponer una solución mejor, desarrollar las soluciones propuestas y hacer mediciones de índices de calidad, comparar y analizar las nuevas soluciones [1]. Es una metodología iterativa donde en cada iteración la solución se refina hasta encontrar la más apropiada, recordando las etapas de cada iteración a las fases del clásico método científico: Búsqueda de problema, formulación de hipótesis, predicción y validación de hipótesis.

1. *Observar las soluciones existentes.*

   En la fase exploratoria se estudiará la literatura y el estado del arte del campo de investigación tratando de determinar posibles mejoras.

2. *Proponer soluciones mejores.*

   Esta fase está dedicada al análisis y diseño de soluciones mejores, tratando de superar los límites de las propuestas previas o mejorando los métodos existentes en la literatura.

3. *Desarrollar la solución.*

   El objetivo de esta fase es desarrollar un prototipo para demostrar las características de nuestra solución.

4. *Medir y analizar la nueva solución.*

   Por último, los prototipos implementados son empíricamente evaluados y comparados con las diferentes alternativas. El objetivo de esta evaluación es validar la nueva solución propuesta y corroborar que los problemas descubiertos en la primera fase fueron resueltos.

R.2.2 Objetivos

Hemos definido dos tareas principales con el fin de resolver la pregunta de investigación. Cada tarea se trata en una de las dos partes en las que se organiza esta Tesis Doctoral, y propone diferentes objetivos o items a cumplir para resolver el problema de investigación global considerado (ver Fig. 1). La metodología de investigación descrita se aplicará a cada objetivo.

Trabajo previo

En los últimos años el grupo de investigación Trasgo [61] ha diseñado y desarrollado Hitmap [48, 55], una biblioteca altamente eficiente para el particionado y mapeado jerárquico en memoria distribuida de estructuras de datos indexadas como vectores, matrices multidimensionales, matrices dispersas o grafos. Hitmap esta diseñada para utilizar una abstracción que permita simultáneamente una visión global o local de una computación paralela, permitiendo la creación, manipulación, distribución y eficiente comunicación de estructuras de datos *teseladas* (particionadas) de forma abstracta. El programador no necesita pensar en el número de procesos y procesadores. En su lugar, Hitmap usa una abstracción para la construcción de

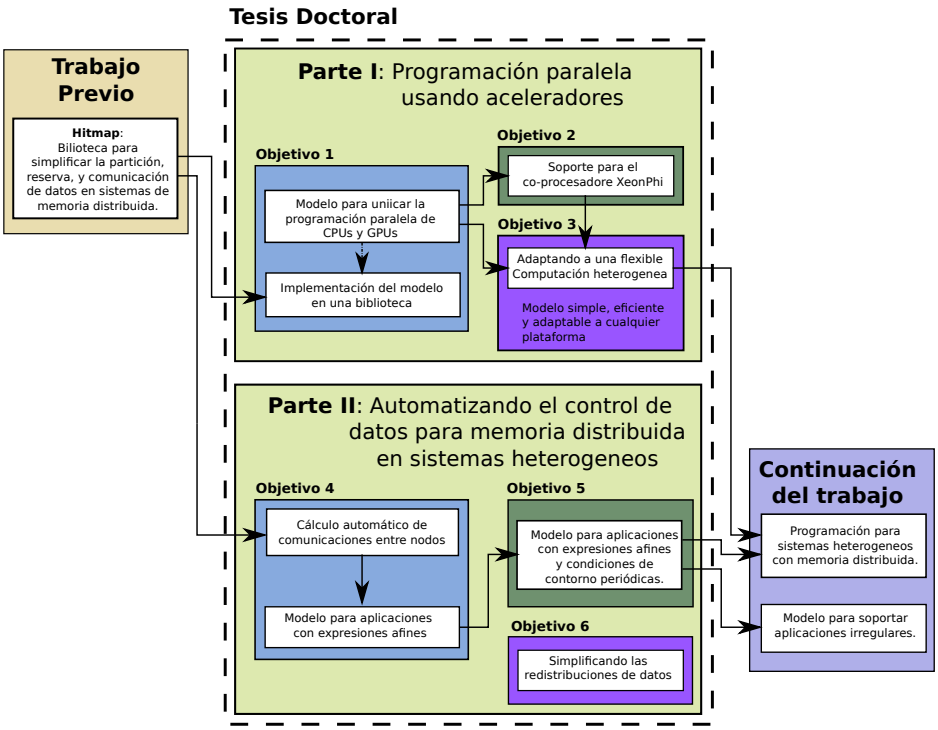

Figura 1: Objetivos y estructura de la Tesis Doctoral.

patrones de comunicación para las estructuras distribuidas de cualquier nivel, con estados globales claramente identificados. Así, las operaciones de codificación y verificación son mucho más fáciles.

Esta biblioteca da soporte a funcionalidades de: (1) Organización de los elementos de proceso en topologías virtuales; (2) Partición y asignación de partes de las estructuras de datos a los diferentes procesos usando diferentes técnicas de equilibrio de carga de forma modular; (3) Determinación y gestión automática de los procesos inactivos en cualquier estado de la computación; (4) Identificación de los procesos vecinos o relacionados para mejorar las comunicaciones en su caso; y (5) Construcción abstracta de patrones de comunicación, que además son reutilizables en algoritmos iterativos.

Parte I: Simplificando la programación de sistemas heterogéneos basados en aceleradores

En la primera parte de la Tesis Doctoral presentaremos un modelo de programación que simplifica la programación en paralelo de sistemas heterogéneos, entendiendo un sistema

heterogéneo como aquel sistema compuesto por un host (proceso principal) y diferentes dispositivos de computación asociados al host (grupos de núcleos de CPU, o aceleradores del tipo GPUs o XeonPhi).

Objetivo 1: Diseñar y desarrollar un modelo de programación para unificar la programación de CPUs y GPUs.

(*Observación*) Actualmente el uso de aceleradores hardware, como las unidades de procesamiento gráfico (GPUs), es clave para resolver problemas de alto coste computacional que requieren HPC. Sin embargo, programar soluciones para un eficiente despliegue en este tipo de dispositivos es una tarea muy difícil, que requiere del tratamiento manual de las transferencias de memoria y de los parámetros de configuración. El programador es el responsable de realizar un estudio de los datos necesarios en cada momento para la computación en cada diferente coprocesador, teniendo en cuenta a su vez los detalles de la arquitectura de ejecución.

(*Propuesta e implementación*) Proponemos el concepto de *Controller*, una entidad abstracta que permite ocultar los detalles de las comunicaciones y lanzamientos de kernels en los aceleradores hardware, facilitando su programación. El modelo también ofrece la posibilidad de definir y lanzar kernels para CPU en procesadores multi-core, usando la misma abstracción y metodología que para las GPUs.

(*Resultados*) Presentaremos una implementación del modelo de los Controladores en un prototipo construido como una biblioteca de funciones, junto con una serie de casos de estudio. Demostraremos que el uso de esta biblioteca reduce el esfuerzo de programación y los costes de portar el código entre dispositivos, sin añadir sobrecostes en los tiempos de ejecución cuando se comparan con soluciones directamente programadas y optimizadas con CUDA u OpenMP.

Objetivo 2: Soporte para un nuevo tipo de aceleradores como el coprocesador Xeon Phi.

(*Observación*) Soportar diferentes tipos de dispositivos como las GPUs o las Xeon Phis en los modelos de programación actuales es vital para explotar los sistemas paralelos. Diferentes tipos y familias de aceleradores son usados en las arquitecturas modernas de cómputo, como podemos observar en la configuración de los los supercomputadores de la lista TOP500 [130]. Sin embargo, los modelos de programación usados típicamente para GPUs y Xeon Phis son muy diferentes.

(*Propuesta e implementación*) En este punto proponemos una extensión del modelo de programación heterogéneo anterior para GPUs y CPUs, obteniendo una nueva aproximación a la programación homogénea de CPUS, GPUs y XeonPhis. Esta contribución extiende el modelo de los Controladores anterior y su implementación, combinando el modelo de comunicaciones diseñado para GPUs con el modelo de ejecución diseñado para CPUs. Esto resulta en una nueva solución para dar soporte a la Xeon Phi.

(*Resultados*) Los resultados experimentales de esta parte muestran que usando la nueva solución, el esfuerzo de programación necesario para cambiar la plataforma de ejecución ha sido muy reducido en muchos casos de estudio. Por ejemplo, usando nuestro modelo para programar el benchmark Mandelbrot, el 97 % del código es reutilizado entre la implementación para GPU y la de Xeon Phi.

Objetivo 3: Diseñar y desarrollar un modelo de programación adaptable, simple y eficiente para la programación de sistemas con múltiples dispositivos heterogéneos.

(*Observación*) Existen muchas propuestas para simplificar la programación, el manejo de varios aceleradores, y la programación de sistemas híbridos mezclando aceleradores y CPUs. Sin embargo, la portabilidad muchas veces compromete la eficiencia de los programas en los diferentes dispositivos. Además, hay detalles acerca de la coordinación y el manejo de memoria que deben ser todavía atacados por el programador.

(*Propuesta e implementación*) En este objetivo proponemos la entidad *Multi-Controlador* (*MCtrl*), una entidad abstracta implementada también en una biblioteca. Esta entidad coordina los diferentes dispositivos (incluyendo aceleradores o grupos de CPUs) que se pueden encontrar en un sistema heterogéneo. Nuestra propuesta usa el modelo de Controladores y la biblioteca Hitmap para crear un nuevo modelo de programación que mejora las soluciones del estado del arte, simplificando la partición y mapeo de datos y haciendo transparente la computación de los kernels en los diferentes dispositivos. El sistema runtime selecciona y ejecuta automáticamente la mejor implementación de kernel existente (entre las suministradas por el programador) para cada dispositivo, haciéndose cargo también de los movimientos necesarios de datos y ocultando los detalles del lanzamiento de los kernels en los diferentes dispositivos.

(*Resultados*) Los resultados de un estudio experimental con varios casos de estudio indican que nuestra abstracción permite el desarrollo de programas flexibles y eficientes en múltiples dispositivos o unidades computacionales, que se adaptan a los entornos heterogéneos.

Al final de esta parte de la Tesis Doctoral obtendremos un modelo de programación que simplifica la programación en paralelo sobre sistemas heterogéneos, logrando una gran eficiencia en términos de tiempo de ejecución para aplicaciones que no tengan dependencias de datos.

El soporte de aplicaciones con dependencias de datos en sistemas que incluyen dispositivos con espacios de memoria separados, requiere el diseño y desarrollo de técnicas para realizar automáticamente las transferencias de datos entre dispositivos. Este tipo de técnicas son presentadas en la parte II.

Parte II: Automatizando el manejo de datos en sistemas heterogéneos con memoria distribuida.

Los casos más significativos donde los dispositivos tienen un espacio de memoria separado son los sistemas distribuidos. En esta parte de la Tesis Doctoral presentamos un conjunto de técnicas genéricas y automáticas que simplifican la programación en paralelo sobre sistemas de memoria distribuida. Estas hacen transparente al programador muchos de los problemas relacionados con la partición, reserva y transferencia de datos.

Objetivo 4: Diseño y desarrollo de una técnica en tiempo de ejecución que calcula automáticamente las comunicaciones agregadas para las aplicaciones con expresiones afines y uniformes en el acceso a datos.

(*Observación*) La programación de aplicaciones en sistemas distribuidos es un verdadero reto para los programadores. Éstos tienen que tratar con muchas decisiones que no están relacionadas con el algoritmo, como son las decisiones acerca de la partición de datos, localidad, costes de comunicación, sincronización, etc. Las técnicas actuales de compilación no pueden tener en cuenta muchas decisiones que están basadas en información global y particular de los sistemas de ejecución y sus dispositivos asociados. Por ello, es deseable estudiar qué tipo de técnicas pueden ser aplicadas en tiempo de ejecución, con el fin de tomar esas decisiones lo más adecuadamente posible.

(*Propuesta e implementación*) Presentamos una nueva técnica para el cálculo de las comunicaciones de manera automática. La técnica es aplicada entre diferentes bloques SPMD (Single Program Multiple Data) y puede ser usada en códigos con expresiones de acceso a datos afines y uniformes. La técnica propuesta calcula en tiempo de ejecución las comunicaciones agregadas y exactas necesarias para procesos distribuidos, haciendo transparente al programador los problemas relacionados con el código de las comunicaciones.

(*Resultados*) Los resultados experimentales muestran que, a pesar de el coste potencial de nuestro cálculo en tiempo de ejecución, nuestra solución puede producir automáticamente códigos eficientes comparados con programas desarrollados directamente en MPI, y comparados también con los códigos generados por los compiladores autoparalelizadores del estado del arte.

Objetivo 5: Extensión de la técnica anterior para dar soporte a aplicaciones con dominios periódicos.

(*Observación*) Hay muchas aplicaciones reales que usan accesos a datos en dominios periódicos. Así, cada vez es más interesante dar soporte a este tipo de aplicaciones en los entornos de programación paralelos o compiladores actuales.

(*Propuesta e implementación*) Hemos realizado una extension de la técnica de cálculo de comunicaciones anterior, con el fin de soportar códigos de entrada con expresiones de acceso a datos uniformes con condiciones de contorno periódicas.

(*Resultados*) Hemos evaluado nuestra propuesta usando varios casos de estudio. Los resultados experimentales muestran que el uso de nuestra solución puede obtener automáticamente códigos eficientes comparada con implementaciones directamente desarrolladas en MPI. Además, nuestra solución simplifica la programación eliminando la necesidad de desarrollar el código de comunicaciones y partición de datos.

Objetivo 6: Proposición de una abstracción para simplificar las redistribuciones de datos.

(*Observación*) La programación en sistemas distribuidos impone una serie de retos específicos. Una de las optimizaciones más compleja de desarrollar de forma genérica, con propensión a introducir errores en la programación, y sin embargo más útil en sistemas distribuidos, es la redistribución de datos. Esta optimización cambia la afinidad de los datos seleccionados, moviéndolos a las nuevas localizaciones que son determinadas por unas técnicas de partición y mapeo diferentes de las originales. Las redistribuciones de datos permiten una mejora en el rendimiento gracias a la creación de un mejor equilibrio de carga entre los procesos que se encuentren activos en la computación cada momento.

(*Propuesta e implementación*) Nuestra propuesta se basa en el diseño de cuatro operadores para redistribuir una selección de datos en memoria distribuida de una forma eficiente y simple. En nuestra propuesta la partición, recolocación y movimiento de datos es transparente para el programador. Los operadores abstraen al programador los detalles de implementación que son dependientes de la máquina de ejecución como por ejemplo el número de procesos activos donde los datos están localizados antes y después de la redistribución. La combinación de estos cuatro operadores permite un simple y eficiente desarrollo de muchas estructuras de aplicación, incluyendo muchos patrones de paralelismo que encontramos en la biblioteca C++ STL.

(*Resultados*) Los resultados experimentales muestran que nuestra solución no implica sobrecostes de ejecución comparada con las redistribuciones manualmente escritas en MPI, mientras el esfuerzo de programación ha sido altamente reducido.

Al final de la segunda parte de la Tesis Doctoral obtendremos un conjunto de técnicas que hacen transparente al programador muchos de los problemas relacionados con la partición de datos y la transferencia de estos entre dispositivos, en aplicaciones diseñadas para ejecutarse en sistemas con espacios de datos disjuntos o distribuidos.

Continuación del trabajo

Una continuación natural de las propuestas incluidas en este trabajo de tesis, es unificar las diferentes técnicas en un modelo de programación para memoria distribuida simple y adaptable a los sistemas heterogéneos. Proponemos para ello la combinación del modelo de programación presentado en el final de la primera parte de la Tesis Doctoral y las técnicas

presentadas en la segunda parte para tratar automáticamente con las transferencias de datos entre los dispositivos encontrados en las máquinas heterogéneas. Así, la plataforma de ejecución podría ser un sistema de memoria distribuida, donde cada nodo podría tener varios procesadores y coprocesadores asociados (e.g GPUs, XeonPhi, o grupos de núcleos de CPU).

R.3 Resumen de contribuciones

Esta sección resume las contribuciones de esta Tesis Doctoral, y las publicaciones asociadas a ella.

R.3.1 Respuesta a la pregunta de investigación y conclusiones

Durante la Tesis Doctoral presentaremos, de acuerdo a los objetivos marcados, una serie de contribuciones que nos llevarán finalmente a la respuesta de la pregunta de investigación. En este caso la respuesta será afirmativa, ya que presentaremos: (1) Un modelo de programación simple y adaptable a los diferentes sistemas heterogéneos de ejecución compuestos por diversos aceleradores, y (2) varias técnicas automáticas capaces de realizar las transferencias de datos necesarias en sistemas de memoria distribuida o con espacios de memoria separados.

Nuestras soluciones han llevado a la publicación de varios artículos de investigación como veremos en esta sección. El resumen de las contribuciones está dividido en dos partes, de forma similar a la estructura de la Tesis Doctoral y a la pregunta de investigación.

R.3.2 Simplificando la programación sobre sistemas heterogéneos basados en aceleradores

La programación de soluciones para una eficiente ejecución de los programas en un sistema heterogéneo es una tarea muy compleja, donde el programador debe manualmente considerar los detalles de la arquitectura de la máquina. En la primera parte de la Tesis Doctoral, presentaremos varias propuestas para reducir este problema. Las publicaciones derivadas de este trabajo de investigación son las siguientes:

5. Presentamos una nueva entidad abstracta denominada *Controllers* implementada como una biblioteca de funciones. Esta entidad permite la creación de códigos portables de coordinación para diferentes dispositivos como GPUs o grupos de núcleos de CPU. Esta solución también permite la programación de kernels simples y genéricos que pueden ser ejecutados en ambos tipos de dispositivos, y hace transparente al programador los movimientos de datos entre las diferentes jerarquías de memoria.
   Esta solución cumple el Objetivo 1: Diseñar y desarrollar un modelo de programación para unificar la programación de CPUs y GPUs.
   Publicación:

Journal JCR Q2: [99] Ana Moreton–Fernandez, Hector Ortega–Arranz y Arturo Gonzalez–Escribano. 'Controllers: An abstraction to ease the use of hardware accelerators'. En: *The International Journal of High Performance Computing Applications*, 2017. 2017. DOI: 10.1177/1094342017702962. eprint: http://dx.doi.org/10.1177/1094342017702962. URL: http://dx.doi.org/10.1177/1094342017702962

6. Introducimos en el modelo anterior de *Controllers* el soporte para un nuevo tipo de dispositivo, el coprocesador Intel Xeon Phi (MIC). Se ha introducido sin realizar ninguna modificación semántica del modelo anterior. Para ello, se ha unido el modelo de ejecución usado para los grupos de núcleos de CPU y el modelo de comunicaciones usado para las GPUs, mientras su implementación se ha realizado con el lenguaje específico para la Xeon Phi.
   Con ello logramos el **Objetivo 2:** Soporte para un nuevo tipo de aceleradores como el coprocesador Xeon Phi.
   Publicación:

   Conference CORE A: [100] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano y Diego R Llanos. 'Supporting the Xeon Phi Coprocessor in a Heterogeneous Programming Model'. En: *European Conference on Parallel Processing*. Springer, Cham. 2017, págs. 457-469

7. En este punto proponemos una entidad abstracta que es capaz de coordinar de manera transparente las operaciones realizadas por varios Controllers sobre diferentes dispositivos de cómputo, sin importar el tipo de dispositivo que el Controller tenga asociado.
   Con ello cumplimos el **Objetivo 3:** Diseñar y desarrollar un modelo de programación adaptable, simple y eficiente para la programación de sistemas heterogéneos.
   Publicación:

   Journal JCR Q3: [96] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Multi-device Controllers: A library to Simplify Parallel Heterogeneous Programming'. En: *International Journal of Parallel Programming*, 2017, págs. 1-20. Springer, 2017

R.3.3 Automatizando el manejo de datos en sistemas heterogéneos con memoria distribuida.

La programación paralela de aplicaciones con dependencias de datos ha sido un gran reto para los programadores, especialmente en sistemas con las memoria separada, donde este tipo de aplicaciones implica comunicaciones de datos. En la segunda parte, se han tratado diferentes problemas relacionados con los sistemas de memoria distribuida. Nuestras propuestas, basadas en el movimiento de técnicas de compilación a tiempo de ejecución,

mejoran las propuestas encontradas en la literatura ya que: son independientes de decisiones en tiempo de compilación, generan comunicaciones exactas y de grano grueso o producen un eficiente equilibrio de carga de trabajo.

Las publicaciones derivadas de este trabajo de investigación son las siguientes:

8. Estudio de los costes de tiempo de ejecución producidos por los códigos generados automáticamente en los sistemas de memoria distribuida por las herramientas del estado del arte basadas en el polyhedral model. Este estudio analiza las limitaciones de las técnicas actuales y propone una simplificación que elimina un factor de complejidad.
   Publicación:

   Conference CORE B: [97] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano y Diego R Llanos. 'On the run-time cost of distributed-memory communications generated using the polyhedral model'. En: *High Performance Computing & Simulation (HPCS), 2015 International Conference on*. IEEE. 2015, págs. 151-159

9. Desarrollamos una herramienta básica de programación para la comprobación de nuevas técnicas de programación. Se tratará de un framework de programación que simplifica el desarrollo de códigos paralelos y la extracción automática de las dependencias de datos.
   Publicaciones:

   International Conference: [8] Arturo Gonzalez-Escribano Ana Moreton-Fernandez y Diego R. Llanos. 'Trasgo 2.0: Code generation for parallel distributed- and shared-memory hierarchical systems'. En: *Compilers for Parallel Computing (CPC)*. London, 2015

   International Conference: [93] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano y Diego R Llanos. 'A New High-Level Parallel Portable Language for Hierarchical Systems in Trasgo'. En: *Computational and Mathematical Methods in Science and Engineering (CMMSE)*. 2015

10. Presentamos una nueva técnica capaz de calcular en tiempo de ejecución las comunicaciones necesarias para ejecutar programas de forma paralela sin variar la semántica del código. Esta técnica es aplicable a aquellos trozos de código que contengan expresiones en los accesos a datos afines y uniformes.
    Con esta técnica cumplimos el **Objetivo 4:** Diseñar y desarrollar un técnica de runtime capaz de calcular automáticamente las comunicaciones necesarias, de forma agregada, en códigos con expresiones afines y uniformes.
    Publicaciones:

    Poster International School: [7] Arturo Gonzalez-Escribano Ana Moreton-Fernandez y Diego R. Llanos. 'Simple and Efficiente parallel programming for distributed-memory systems'. En: *Advanced Computer Architecture and*

Compilation for High-Performance and Embedded Systems (ACACES). Fiuggy, Italy, 2016

Journal JCR Q2: [94] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano y Diego R Llanos. 'A Technique to Automatically Determine Ad-hoc Communication Patterns at Runtime'. En: *Parallel Computing*, 2017. North-Holland, 2017

11. Basándonos en los conceptos de la técnica anterior, desarrollamos una nueva técnica para el cálculo de comunicaciones en tiempo de ejecución capaz de trozos de código con expresiones periódicas en los accesos afines y uniformes a los datos.
Con ello cumplimos el **Objetivo 5:** Extensión de la técnica anterior para dar soporte a aplicaciones con dominios periódicos.
Publicaciones:

    Poster Conference CORE A: [90] Ana Moreton, Arturo Gonzalez-Escribano y Diego R. Llanos. 'A Runtime Analysis for Communication Calculation'. En: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. poster, 2017

    Conference CORE C: [91] Ana Moreton-Fernandez y Arturo Gonzalez-Escribano. 'Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions'. En: *10th International Symposium on High-Level Parallel Programming and Applications (HLPP)*. Valladolid, Spain, 2017

    Journal JCR Q3: [92] Ana Moreton-Fernandez y Arturo Gonzalez-Escribano. 'Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions'. En: *Concurrency and Computation: Practice and Experience*, 2018. Wiley, 2018

12. Finalmente, hemos atacado un problema importante para los sistemas de memoria distribuida cuando se tratan aplicaciones que cambian dinámicamente su carga computacional. Proponemos cuatro nuevos operadores que pueden ser usados para rebalancear la carga de múltiples tipos de aplicaciones.
Estos operadores tienen la finalidad de cumplir el **Objetivo 6:** Proposición de una abstracción para simplificar las redistribuciones de datos.
Publicaciones:

    Conference CORE C: [6] Arturo Gonzalez-Escribano Ana Moreton-Fernandez y Diego R. Llanos. 'Four abstract array distribution operators'. En: *9th International Symposium on High-Level Parallel Programming and Applications (HLPP)*. Muster, Germany, 2016

    Journal JCR Q2: [98] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Operators for Data Redistribution on the STL library

and RayTracing Algorithm (Submitted)'. En: *Journal of Parallel and Distributed Computing*, 2018. Elsevier, 2018

R.4 Conclusiones

Esta Tesis Doctoral trata varios problemas importantes relacionados con la programación en paralelo de los sistemas heterogéneos. Las contribuciones de la primera parte de la Tesis Doctoral habilitan la creación de estructuras de programación que permiten la programación de código de coordinación portable entre diferentes dispositivos, aceleradores y arquitecturas, manteniendo a su vez una eficiencia máxima gracias al uso interno de herramientas de bajo nivel y específicas de los proveedores. En la segunda parte, se ha tratado con muchos de los problemas relacionados con los sistemas de memoria distribuida. Nuestras propuestas, basadas en el movimiento de técnicas de compilación a el tiempo de ejecución, abren la puerta a un futuro soporte de un mayor rango de aplicaciones en los futuros compiladores o entornos de programación.

La combinación de ambas contribuciones conllevará una nueva solución para sistemas heterogéneos con varios nodos distribuidos. Será posible gracias a la integración de las técnicas realizadas en la parte dos de la Tesis Doctoral, sobre el modelo de programación propuesto en la primera parte.

Chapter 1

Introduction

The use of parallel computing systems frequently represents the only scalable way to solve HPC (High performance Computing) problems in reasonable execution times. The current trend in high performance computing platforms is to include in the same machine several parallel devices, of different type and architectures, and to interconnect them to form highly parallel and heterogeneous distributed systems. Programming efficient and portable parallel applications that can really exploit these systems, imposes specific and complex challenges to the programmers.

This chapter discusses some of these challenges, mainly focused on the portability of programs across different device types, and the coordination of devices with disjoint or distributed memory spaces. We identify problems that lead to the objectives and the research question proposed in this Thesis.

1.1 Motivation

This section presents the motivation of the Thesis, discussing the evolution of parallel systems and the need for new programming models and runtime systems to control them. We will also introduce the work of the Trasgo research group in this particular topic and the challenges that this Thesis aims to solve.

1.1.1 Parallel computing

Many computing problems such as medical simulations, networks algorithms, or autonomous car navigation require complex simulations with high precision levels or an answer to the problem in a limited and generally short time [9]. These requirements imply the execution of programs with a high computational cost, making prohibitive their sequential execution.

A solution for reducing the execution time of these applications can be the increase of the execution-system clock frequency, and hence the processor performance. However, the single processor evolution is reaching a frequency limit. The generated heat when clock frequency increases, leads to excessive power consumption and dissipation problems. This phenomena has stalled the development of the speed in single processor units [47].

The division of the works in several independent tasks, where each processing element of a machine is capable of executing a task independently, can provide a solution for this kind of high-computational problems [3]. More processing units in an execution platform imply that more tasks can be created (being simultaneously executed), and the time spent in the computation will be reduced, achieving in this way a scalable solution for the current applications that are continuously increasing its computational load. This feature nowadays makes the parallel systems the base of the high-performance computing (HPC).

The need to provide a solution to solve HPC problems has pushed forward the development of parallel systems, thus increasing during the last decades the number of processing elements available in the execution systems. On the other hand, the inherent complexity of specifying and coordinating concurrent tasks, the lack of portable algorithms and software development toolkits, and the amount of different kinds of parallel platforms have increased the time and effort needed by the programmer to develop parallel software [56].

In this section, first we describe the main-trend architectures used for parallel computing. After that, we provide a description of the most-known parallel programming models used for each architecture.

1.1.2 Machines for parallel computing

Parallel computing frequently represents the only scalable way to solve problems in reasonable execution times. For this reason, computing evolution has been focused on the design of new computer architectures, instead of the development of faster processors.

The *multi-core* term refers to the integration of several processing units in a single chip, where each one of the processing units operates as an all-purpose processor. The use of several processing units at the same time implies an increase of the performance, without rising the clock speed, as long as the workload can be efficiently distributed among the processing units. The processing elements, according to the processor design, can share memory space, forming a global memory space. The presence of a global memory space make programming such platforms more natural. The read-only memory operations are invisible to the programmer, as they are coded similarly to sequential programs. However, write operations are harder, as they may require mutual exclusion for concurrent accesses, or read-write synchronizations, analyzed and hard-wired in the code by the programmer.

On the other hand, parallel systems which do not share memory spaces present other challenges to the programmer. Each processing node of the platform has its own exclusive address space, so all the interactions among the different nodes should be performed by using communication abstractions such as messages. These messages are used to transfer data, work, or to synchronize actions among the processes.

Co-processors for parallel computing

A co-processor can be considered as a device added to a computing system dedicated to the execution of specialized software. The most representative co-processors for supercomputing nowadays are the GPUs [133]. They were originally designed to help the CPU in graphic processing. In graphic processing algorithms the data reuse is small and the programs are relatively simple. Thus, the design of the GPUs was based on the use of small on-chip caches along with a collection of simple ALUs. The use of co-processors along with the general-purpose processors, has been a main trend in supercomputing since Nvidia launched its Tesla GPU (*Graphics Processing Unit*) card in 2007. GPUs [133] were originally designed to help the CPU in graphic processing, but its powerful computational features made their use very popular in high performance computing, creating the trend of GPGPU (General-Purpose Computing on Graphics Processing Units).

New accelerator devices have arisen in the last years to tackle efficiently different kinds of problems that do not suit perfectly with the GPU architectures. An example is the Intel Xeon Phi (released in 2012), also known as Many Integrated Cores (MICs) [73]. The Intel Xeon Phi coprocessor is an accelerator with many general-purpose cores, based on x86 technology, with improved vector units and memory bandwidth, that can be programmed as a co-processor, or with distributed parallel programming frameworks.

Heterogeneous computing: Taking advantage of all the kinds of machines together

The evolution of supercomputers has been recorded by the Top500 project [130], which ranks the 500 most powerful computer systems in the world. Current supercomputers are composed by several nodes with distributed memory, where each node can have different

computing capabilities, memory hierarchies and co-processor devices associated, such as GPUs or Xeon Phis. In many cases, the computation capabilities associated to each node or different computational units inside a node are highly different, leading to what we know as *heterogeneous systems* [26].

1.1.3 Parallel programming models

According to Balaji [12], a programming model can be defined "as the abstract machine for which a programmer is writing instructions". Typically, these programming models are instantiated in languages or libraries.

With the coming of the parallel computing era, new programming models, that suit well for high-performance parallel computing and supercomputing systems, are being designed by computer science researchers. Programming parallel systems is complicated because the programmer should reason taking into account the stochastic situations and behaviors associated to the simultaneous computation of multiple processing units and data movements, to achieve a correct execution of the program. For this reason, parallel programming models usually include a model, defining the path that the code execution takes, named *execution model*, and the *memory model*, the method determining how data move in the system between the computing nodes and the memory hierarchies of each computing unit.

There are four pillars of programming which represent the possible programmer preferences in the features of a programming model: (1) *Productive*, capable of expressing abstract algorithms with ease; (2) *Portable*, capable of being used on any computer architecture; (3) *Performant*, capable of delivering performance commensurate with that of the underlying hardware; and (4) *Expressive*, capable of expressing a broad range of algorithms. It would be ideal to design a programming languages which would achieve the four characteristics, however it is nearly impossible. This situation leads to a big amount of programming models, each one focused in providing different features. This behavior is not different for the parallel systems. Different users prefer different levels of abstraction and different sets of tradeoffs among the four pillars of programming. In this section, we provide and overview of the most-known parallel programming models used on high-performance computing and supercomputing systems nowadays.

Classification of parallel programming models

Many programming languages and models for parallel computing have arisen for both shared- and distributed-memory systems, with the goal of abstracting to the programmer many implementation issues. OpenMP and the message-passing paradigm (implemented for example by MPI libraries) are the most extended parallel frameworks for shared- and distributed-memory respectively [32, 59]. Both approaches are defined as the *pure parallel models*, as they are dominating the HPC parallel programming landscape since the late 1990s [44].

OpenMP [33, 40] is an application programming interface (API) that eases the parallel programming for shared-memory systems. It is compounded by a set of directives, pragmas, library functions, and a runtime system able to manage the creation of threads, the synchronization operations, and the memory interactions. Originally, OpenMP was designed to parallelize sequential codes with low programmer effort. Sequential codes are annotated with pragmas, and a specialized compiler with support for OpenMP is in charge of the management of the rest of the parallel issues. Nowadays, OpenMP provides new parallel programming structures oriented to dynamic tasks and other more complex parallel structures [10].

There are other models targeting shared-memory systems such as POSIX Pthreads [29, 78]. It is a set of programming language types and procedure calls, with bindings in the most popular programming languages, like for example C/C++, Fortan, Python, etc. The PThreads library provides functions for creating and destroying threads and for coordinating thread activities. However, compared to OpenMP, the Pthreads interface is much lower-level, needing more effort in the code development and maintenance.

Programming for distributed-memory systems imposes specific challenges. Message Passing is the more widespread parallel programming model for distributed-memory systems. All the needed information for the local execution contained in a remote process should be send to the local process by interchanging messages. The MPI standard has dominated this model, being currently the *de facto* standard for HPC applications on distributed architectures [58, 59, 89]. However, using directly the MPI specification, the programmer still has to deal with many implementation issues, such as decisions about partition and locality vs. synchronization/communication costs, or scheduling details. Not only tries MPI to deal with distributed-memory programming challenges, but also low-level networking layers, such as GASnet [20], provide network-independent, high-performance communication primitives tailored for implementing parallel global address space SPMD languages (PGAS) and libraries, with the goal of simplifying the distributed-memory parallel programming to the end users.

Programming a co-processor differs from the multi-core and distributed-memory programming in many aspects. The programmer has to deal with many details such as the manual management of memory transfers to the accelerator device, or the configuration of running parameters that are new for these platforms. For this reason, NVIDIA released in 2007 CUDA [102], a new programming model especially designed for the NVIDIA-GPU accelerators. CUDA programming model is only valid for NVIDIA-GPU accelerators. As for other kinds of co-processors, other programming models for generic GPUs have been proposed such as OpenCL [127] or BrookGPU [27]. Also, for XeonPhi co-processors, LEO [101] offers a similar programming framework.

Parallel programming for multilevel heterogeneous systems

Different programming models have been settled for each different kind of computation architecture, as we described previously. In order to be able to take advantage of the current heterogeneous parallel systems, a programmer must be proficient in distributed-memory

communication tools or layers (e.g MPI, GASNet [20], etc), shared-memory programming models (e.g OpenMP), and specific programming models for the available co-processors (e.g CUDA, OpenCL, or equivalent technologies), for creating hybrid programs that will exploit all the machine capabilities [141]. Moreover, the end user also has to deal with the proper workload distribution among the different nodes and devices, assigning to each one an amount of workload related to their computation power and features.

Nowadays, all these issues should be solved by the programmer, turning the programming of the heterogeneous platforms in an actual challenge.

1.2 Objectives of this Thesis

According to the identified problems described in the previous section, the research question to be solved in this Ph.D Thesis is the following:

> *It is possible to introduce: (1) Simple abstractions to make transparent and uniform the program deployment and coordination on parallel computational units of different types and architectures; and (2) new runtime techniques that take into account the data dependences in order to automatically coordinate the data communication across disjoint or distributed memory spaces, in a transparent way for the programmer, once the features of the execution machine are known?*

1.2.1 Research methodology

The research methodology used in this Thesis is based on the experimental method and a software engineering method that has four different stages: Observe existing solution, propose better solutions, build or develop the proposed solutions, and measure and analyze the proposal [1]. It is an iterative methodology that is repeated to refine the solutions. It resembles the stages of the classical scientific method: Propose a question, formulate hypothesis, make predictions, and validate hypothesis.

1. *Observe existing solutions.*

   This is an exploratory phase where the literature and the state-of-the-art tools related to our research field should be studied in order to determine possible improvements.

2. *Propose better solutions.*

   This phase is dedicated to the design and analysis of better solutions, trying to overcome the limits of previous proposals or to improve methods of the literature.

3. *Build or develop the solution.*

   In this phase, we focus on building a prototype to demonstrate the feasibility of the solution.

4. *Measure and analyze the new solution.*

   Finally, the implemented prototypes are empirically evaluated and compared with different alternatives. The goal of this evaluation is to validate the new solution proposed, and to corroborate that the problems discovered in the first step have been solved.

1.2.2 Milestones

In order to be able to answer the research question, we define two main huge tasks. Each one of these tasks is presented in a part of the Thesis, and contains different goals/items in order to solve the considered research problem (see Fig. 1.1). The research methodology previously described has been applied on each goal.

Previous Work:

During the last years, the Trasgo group [61] has designed and developed Hitmap [48, 55], a highly-efficient functions library for hierarchical tiling and mapping, in distributed memory, of indexed data-structures such as multidimensional array, sparse matrices, or graphs. It is designed to simplify the programming of parallel applications using simultaneously a global and local view of the computation. It allows the creation, manipulation, distribution, and efficient communication of tiles and tile hierarchies of the data structures. The programmer does not need to reason in terms of the number of physical processes or processors. Instead, it uses highly abstract communication patterns for the distributed tiles at any grain level. Thus, coding and debugging operations with entire data structures are easy.

The Hitmap library supports functionalities to: (1) Organize the processors in virtual topologies; (2) Part and map the data grids to the different processors using modular partition and load-balancing techniques; (3) Automatically determine and manage inactive processors at any stage of the computation; (4) Identify the neighbor processors to exploit their relations in communications; and (5) Build communication patterns to be reused across algorithm iterations.

Part I: Simplifying the programming on accelerator-based heterogeneous systems

In this first part of the Thesis, we present a programming model that simplifies the parallel programming on heterogeneous systems, defining a heterogeneous system as a system composed by a host (executing a main program) and different computational units (GPUs, XeonPhi, or groups of CPU-cores).

Goal 1: Design and develop a programming model to unify the parallel programming of CPUs and GPUs.

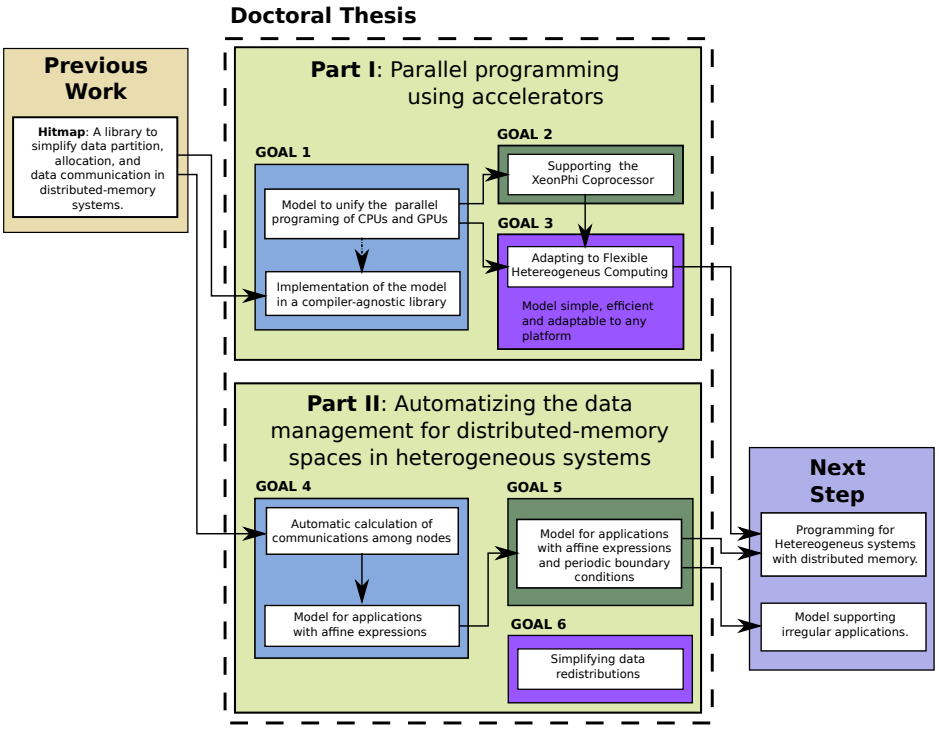

Figure 1.1: Objectives and structure of this Thesis.

(*Observation*) Nowadays the use of hardware accelerators, such as the Graphics Processing Units (GPUs), is key to solve computationally costly problems that require High Performance Computing (HPC). However, programming solutions for an efficient deployment in this kind of devices is a very complex task that relies on the manual management of memory transfers and configuration parameters. The programmer has to carry out a deep study of the particular data needed to be computed at each moment, in different computing platforms, also considering architectural details.

(*Proposal and implementation*) We propose the *Controller* concept as an abstract entity that hides the communications and kernel launching details on hardware accelerators, easing their programming. This model also provides the possibility of defining and launching CPU kernels in multi-core processors with the same abstraction and methodology used for the GPUs accelerators.

(*Results*) We present the implementation of the Controller model in a prototype library, together with its application in several case studies. Its use has led to reductions in the development effort and porting costs, with significantly low overheads in the execution times when compared to manually programmed and optimized solutions using directly CUDA or OpenMP.

Goal 2: Support for new kind of accelerators such as the Xeon Phi Co-processor.

(*Observation*) Supporting computational accelerators such as GPUs or Xeon Phi coprocessors in current programming models is vital to exploit modern parallel platforms. Different kinds and families of accelerators are used in modern high-performance platforms, as we observe in the configuration of the TOP500 supercomputers [130]. However, the usual programming models for GPUs and Xeon Phi platforms are quite different.

(*Proposal and implementation*) We propose an extension of the previous GPU-CPU heterogeneous programming model, to include support for Intel Xeon Phi coprocessors. This contribution extends the previous Controller model and its implementation, by taking advantage of the GPU communication model and the CPU execution model of the original approach, to derive a new approach for the Xeon Phi.

(*Results*) Our experimental results show that using our approach, the programming effort needed for changing the kind of target devices is highly reduced for several study cases. For example, using our model to program a Mandelbrot benchmark, the 97% application code is reused between a GPU implementation and a Xeon Phi implementation.

Goal 3: Design and develop an adaptable, simple, and efficient programming model for multiple heterogeneous devices.

(*Observation*) There are many proposals to simplify the programming and management of several accelerator devices, and the hybrid programming mixing accelerators and CPU cores. However, the portability compromises in many cases the efficiency on different devices. The combined use of different programming models, synchronization and communication techniques, and the details about the coordination and memory management of different types of devices simultaneously, are difficult to be tackled by the programmer.

(*Proposal and implementation*) We propose the *Multi-Controller* (MCtrl), an abstract entity implemented in a library, that coordinates the management of several heterogeneous devices of the same or different type, including accelerators with different capabilities and groups of CPU-cores. Our proposal uses the previous Controller model and the Hitmap library to create a programming model that improves state-of-the-art solutions, simplifying the data partition, mapping, and transparent deployment of kernels. The run-time system automatically selects and deploys the most appropriate implementation of each kernel for each device among the ones provided by the programmer, managing the data movements, and hiding the launching details of kernels distributed across several devices.

(*Results*) Results of an experimental study with several study cases indicate that our abstraction allows the development of flexible and high efficient programs in multiple devices or computational units, that adapt to the heterogeneous environment.

At the end of this first part of the Thesis, we obtain a programming model that simplifies the parallel programming on heterogeneous systems, achieving good efficiency in terms of execution time using the simple methods proposed in the Controller and Multi-Controller abstractions, for applications where there are no data dependences or they are manually solved by the programmer.

The support of applications with data dependences in heterogeneous systems composed by devices with separated memory spaces, requires the design and development of different techniques to automatically perform efficient data transfers among the devices. These kind of techniques are presented in Part II.

Part II: Automatizing the data management for distributed-memory spaces in heterogeneous systems

Coprocessors with disjoint memory spaces are just a particular case of distributed-memory systems. The latencies of data transfers between host and coprocessors, or across distributed nodes, is high in comparison with the computational capabilities of the devices. Thus, minimizing the amount and volume of such transfers and communications is key for performance. In this part we present a set of generic and automatic runtime techniques that simplify the parallel programming on systems with different memory hierarchies or distributed-memory systems in general, making transparent to the programmer all the issues related to the data partition, data allocation, and data transfers.

Goal 4: Design and develop a runtime technique to automatically calculate aggregated communications for applications with uniform affine expressions.

(*Observation*) Programming parallel applications for distributed-memory systems is an actual challenge for developers. The programmer has to deal with many decisions not related with the parallel algorithms, but with implementation issues, such as decisions about partition and locality vs. synchronization/communication costs, scheduling details, etc. Current compile-time techniques cannot take into account many of the tuning decisions that are based on global information and the particular features of the distributed-memory system and its associated devices. It is desirable to study what kinds of techniques can be applied at runtime, in order to deal with the partition and communication issues derived from data-dependences in distributed-memory machines. In particular, we focus on the minimization of the amount and volume of data transfers or communications.

(*Proposal and implementation*) We present a new automatic communication calculation technique to be applied across different SPMD (Single Program Multiple Data) blocks, that can be used in codes with uniform affine expressions for data accesses. The proposed technique computes at runtime exact coarse-grained communications for distributed message-passing processes, making transparent to the programmer the implementation issues related with data communications.

(*Results*) Our experimental results show that, despite the potential cost of our runtime calculation, our approach can automatically produce efficient programs compared with MPI reference codes, and with codes generated with state-of-the-art auto-parallelizing compilers.

Goal 5: Extend the technique to automatically calculate aggregated communications in applications with periodic domains.

(*Observation*) Several real-world applications feature data accesses on periodic domains. Thus, it is increasingly interesting to support these applications on the current parallel programming frameworks or compilers.

(*Proposal and implementation*) We propose an extension of the previous automatic communication calculation technique, in order to support input codes with uniform affine data-access expressions with periodic boundary conditions.

(*Results*) We evaluate our proposal using several study cases. Our experimental results show that the use of our approach can automatically obtain efficient codes when they are compared with pure MPI implementations. Moreover, our approach simplifies the programming by eliminating the need for developing data communication and data partition codes.

Goal 6: Propose an abstraction to simplify data redistributions.

(*Observation*) One of the distributed-memory optimizations which is most difficult to develop in a generic form, which is error-prone, but is highly useful at the same time, is the data redistribution. It changes the ownership or affinity of selected data to processors, moving the data to their corresponding new locations. Data redistributions allow the improvement of performance in many applications by creating a balanced workload among the active processes during the computation evolution. However, there are many different possibilities and conditions that should be taking into account by the programmer to derive a data redistribution strategy for a specific application, some of them related to information only known at run-time.

(*Proposal and implementation*) We propose four operators to redistribute selected data on distributed-memory systems in an efficient and simple way, making the data partition, relocation, and data movement transparent to the programmer. They abstract to the programmer implementation details which are dependent on the execution machine such as the number of active processes where the data are located before and after the redistribution. The combination of these four operators enable to express many common application structures, including parallel patterns found in the C++ STL Library.

(*Results*) Our experimental results show that our approach does not imply significant overheads compared with data redistributions directly written using MPI, while highly reducing the code development effort.

At the end of this second part of the Thesis, we obtain a set of techniques which make transparent to the programmer many of the issues related to the data partition, and the data transfers among different nodes, in applications to be executed on systems with disjoint or distributed-memory spaces.

Continuing Work

We propose as continuing work, the design and development of a simple and adaptable programming model for distributed-memory heterogeneous systems, using as baseline the combination of the programming model presented at the end of the first part, and the techniques presented in the second part, to deal automatically with data transfers among the computational devices found in a heterogeneous machine o cluster. The target execution platform could be a distributed-memory system, where each host will have several computational units associated (e.g GPUs, XeonPhi, or groups of CPU-cores).

1.3 Document structure

This document is structured in two parts. In the first part, Chapter 2 discusses the related work of the first part, in Chapter 3 we present the Controller library (Goal 1), its extension

to support the Xeon Phi co-processor in Chapter 4 (Goal 2), and finally in Chapter 5 a heterogeneous programming model for heterogeneous systems (Goal 3).

In the second part, we describe first in Chapter 6 the work related to this part. Chapter 7 analyses the most sophisticated compilation technique related to our research. In Chapter 8 we introduce a runtime technique to automatically calculate aggregated communications for applications with affine expressions (Goal 4). Chapter 9 redesigns the previous technique to support periodic boundary conditions (Goal 5). Chapter 10 proposes an abstraction to simplify data redistributions (Goal 6). Finally, we present the conclusions and the publications associated to this Thesis.

Part I

Simplifying the programming on accelerator-based heterogeneous systems

In this part of the Thesis we tackle the problems related to the difficulty of generating portable and efficient programs that can be executed on heterogeneous systems, mixing diverse co-processors or accelerators with multi-core and many-core processors. Issues related to the coordination of computations launched in several and/or different devices, transparent memory transfers, and portability of the codes are discussed, and a solution is proposed and studied.

CHAPTER 2

State of the art on heterogeneous programming

In this chapter we expose the different solutions proposed in the community to ease the parallel programming on platforms with accelerators. They cover since completely new programming tools and compilers, to new programming libraries for specific kinds of applications, or new programming methodologies combining several programming models.

2.1 Motivation

In this chapter we discuss the approaches presented in the literature to tackle the difficulty of programming for heterogeneous systems. There are two main problems when programming for heterogeneous systems that are compound by several accelerators of different nature. Thus, we divide the chapter in two sections:

- The first issue came from the different programming methodologies settled for each different kind of computation architecture. In the first section, we study the proposals that try to unify the programming methodology, independently of the composition of the devices of the execution platform, thus easing to the programmer the management of the different computational devices.
- The second issue refers to the complexity related to the distribution of the computation on several different devices at the same time. It includes, among others, the management of the synchronizations, computation distribution, load balancing, or asynchronous data transfers. In the second section, we study the proposals that not only abstract the programming of the execution platform, but also manage the computation distribution, synchronizations, load balancing, or data transfers programming issues among the different computational devices that compound the execution machines.

2.2 Proposals for standardizing parallel programming

In this section we analyse different works proposed in the literature that target the simplification of parallel programming on heterogeneous systems that have different computational devices, including accelerators. These proposals avoid the need for using a manual combination of the specific or vendor-provided programming models for each computation device. They introduce unified programming models or tool abstractions to manage the architectural differences between computational units of different nature or capabilities.

One of these models, widely known, is OpenCL [127]. It provides the *Context* abstraction, defining a memory model in which associated data is shared or moved between the host and the device memories when needed. Although OpenCL is internally exploiting the vendor drivers and native programming tools, its abstractions have been proved to prevent obtaining the same efficiency as when using directly the vendor programming models, for several common situations [76]. Moreover, the implementations are not easily reachable with respect to the definition of the management policies of the internal queues, or the possibility of changing them. Another drawback of this model is the manual management of kernel compilation at run-time, for different architectures in different contexts, that is desired to be generalized and simplified.

Using current heterogeneous code generators or compilers, the code should be recompiled for each different execution platform in order to better exploit the performance capabilities of the system. One example is OpenAcc [139]. It provides a simple and abstract programming framework for accelerators. However, the code should be recompiled with their specific compilers for each different execution architecture in order to achieve a good performance.

The framework presented in [64] allows the development of hybrid MPI+OpenMP programs, generating parallel code depending on the features extracted for the sequential functions. However, it does not support conversions for CUDA.

The llCoMP tool [116] is a source-to-source compiler that translates C annotated code to MPI+OpenMP or CUDA. However, it does not support the joint use of CUDA with the other parallel models. SkelCL [125] enhances the OpenCL interface to allow the coordination of different GPUs on the same machine. The works [37, 77] introduce hybrid MPI+CUDA approaches to coordinate GPUs in the same or different host machines. Apart from their specific limitations, they do not have an abstract support to easily manage homogeneously units of different nature, including CPU cores.

There are other approaches that are more domain specific, but include small abstractions to ease the management between the accelerator and the host (e.g. MCUDA [128] or hiCUDA [65]). They do not consider guided optimizations, nor allow the programmer to control the load distribution or the devices coordination. Other approaches with similar limitations also consider CUDA, MPI, and OpenMP combinations (e.g. [71, 142]).

More general approaches propose complete integrated frameworks, such as OMPICUDA [84], StarPU [72], or the skeleton programming framework based on it, SkePU [41]. In general, they hide the coordination details to the programmer, to the point of constraining the potential optimizations that could be achieved manually. The selection of launching parameters like the thread-block size is tackled in SkePU, but using a trial-and-error method, thus leaving no possibility to extrapolate the results to other kernel codes or architectures. A higher-level approach is to rely on compiler technology to transparently generate code for different kinds of devices (both for coordination and for kernels) from a single unified language. For example, C++14 is used in PACXX [62], a transformation system integrated into the LLVM compiler framework. It transforms explicit parallel constructions that use the concept of kernel and launching in an abstract an elegant form. On the other hand, some of the solutions are dependent on features of this language, and they are not easily portable to other ones. The decisions about launching parameters, such as the thread-block choice, are still the programmer responsibility alone.

Some specific-domain libraries address the portability problem internally using several native programming models. For example, MAGMA library [45] provides a unified programming environment for heterogeneous systems using both CPUs and accelerators, such as GPUs or Intel Xeon Phi, for dense linear algebra algorithms, with complete different implementations for each type of device that cannot be easily used together.

2.3 Proposals targeting directly heterogeneous systems

In this section we discuss the different approaches that, besides of the unification of the programming for different execution platforms, they also abstract to the programmer the coordination of devices of different natures. Thus, they tackle the management of data sharing or partitioning, computation mapping, load balancing, and communication across the different computational units that compound the execution platform, including accelerators.

As we describe in the previous section, OpenCL [127] is a widespread programming framework to deal with heterogeneous devices. However, the coordination of devices of different natures, and the management of data sharing are tricky, and should be manually solved and coded by the programmer. Many libraries of higher level of abstraction such as [43, 57, 69, 79, 137], manage automatically the issues related to the coordination of devices of different natures. However, as these libraries rely on OpenCL as execution layer, they typically inherit the OpenCL problems, being difficult to reach the best possible performance using their solutions (see e.g. [76]).

An interesting example of abstraction built on top of OpenCL is the Maat library [107]. It provides a unified context with an abstract view, regardless of the number and nature of devices, for GPU and CPU platforms. However, it does not allow the exploitation of features of the native programming models, or specialized third-party libraries optimized for the device.

Some of these approaches [17, 107, 136] also automatize the division of the workload inside a node, containing this node several different computational devices. They use an adaptive and dynamic distribution of the iterations of a parallel loop. However, they only support loops with no data dependences.

2.4 Summary

Reviewing the related work, we can affirm that it would be desirable the creation of a new programming model, that, unlike previous approaches, allows the programming with generic and portable kernels, and at the same time, the integration of higher levels of optimization using the native or vendor provided programming models, tools, or libraries, for higher efficiency and performance. In the Chapter 3, we propose a flexible library, and its extension to support Xeon Phi coprocessors in Chapter 4, fulfilling the desired features.

Moreover, in the Chapter 5, we design a library built on native programming models, able to manage in an abstract way to the programmer the issues related to the heterogeneous platforms such as the coordination of the different devices, the data partitioning, or the computation distribution and load balancing.

Chapter 3

Controllers: An abstraction to ease the use of hardware accelerators

The purpose of this chapter is to tackle the first goal of the Thesis:

Design and develop a programming model to unify the parallel programming of CPUs and GPUs.

In order to do that, we introduce the *Controller* programming model. It is based on an abstract entity that allows the programmer to easily manage the communications and kernel launching details on hardware accelerators in a transparent way. This model provides the possibility of defining and launching CPU kernels in multi-core processors with the same abstraction and methodology used for accelerators.

Additionally, the model also allows the programmer to simplify the proper selection of values for several configuration parameters that should be selected when a kernel is launched.

3.1 Motivation

Nowadays the use of hardware accelerators, such as the Graphics Processing Units (GPUs) or XeonPhi coprocessors, is key to solve costly computational problems, that require High Performance Computing (HPC). However, programming solutions for an efficient deployment in this kind of devices is a very complex task that relies on the manual management of memory transfers and configuration parameters. The programmer has to carry out a deep study of the particular data needed to be computed at each moment in different computing platforms, also considering architectural details.

When developing solutions to be deployed in heterogeneous systems, programmers can: (1) Use a single programming model responsible of managing the architectural and conceptual differences between the different computational devices, e.g. OpenACC [39, 139]; or (2) Use a combination of different programming models specific for each kind of computational device, e.g. MPI together with CUDA, OpenCL, or OpenMP.

One of the main drawbacks of the first approach is the difficulty to represent in a generic form non-completely regular programs, with non-trivial communications or synchronizations. Besides, the final generated code resulting from programming with this kind of models is not usually as optimized as a manually developed and tuned code. For example, the current OpenACC compiler implementations do not offer a complete solution to the problem of automatically choose appropriate values for several kernel-launching parameters without any programmer guidance, as required by the standard. These parameters include the thread-block size and its multi-dimensional geometry, or the configuration of the size of the L1 cache vs. the size of the shared-memory in modern GPUs. On the other hand, implementing solutions following the second approach requires a deeper knowledge of the different parallel programming models involved, using different synchronization and memory access strategies for different devices. Additionally, the programmer is the final responsible of properly managing the data transfer among the different memory spaces of the computational devices, at the most appropriate times, together with the choice of proper values for kernel-launching configuration parameters. However, this manual adjustment gives to the programmer the possibility to optimize the use of the particular resources of each specific device. Other approaches that try to create a conceptual bridge between these two approaches are domain specific, or are based in sophisticated compiler technology for the generation of kernel coordination codes.

In this chapter we present the *Controller* library. It unifies the programming for computational devices of very different natures in the context of a compiler library, using a new approach based on simple abstractions. It hides the differences of execution models up to the point of allowing the development of generic kernels portable across devices. But at the same time, it allows the integration of native or vendor programming models, applying specific optimization techniques, and avoiding efficiency losses associated with other generic approaches, solving the limitations discussed in the related work (see Sect. 2.2).

3.2 Controller Model

The *Controller* model [99] introduces a simplified way to program applications that can exploit heterogeneous computational platforms including accelerators or/and multi-core CPUs. The model has several important features: (1) A mechanism to define common kernels reusable across different types of devices, or specialized kernels for specific device kinds, including the possibility to consider a subset of CPU cores as a single independent device; (2) An optimization system to select proper values for kernel-launching configuration parameters (such as the thread-block geometry), guided by simple qualitative code characterization provided by the programmer; (3) A transparent mechanism of memory management, including optimized communications of the data structures between the host and the corresponding images in the accelerators; (4) An abstraction for indexed data structures that unifies the data management in kernels for different kinds of devices (such as GPUs, or multi-threaded vector CPU cores).

The Controller model uses the most appropriate programming models (CUDA, OpenMP, ...) to exploit the computational resources of the accelerators and hosts machines. Its architecture is represented in Fig. 3.1. The Controller coordinates the execution of series of kernels. These kernels are declared as functions, that are managed by the Controller entities. Controllers automatically manage the two main concepts used in a program that exploits accelerators:

- Kernel management, including the kernel launching and configuration. The Controller manages the deployment/execution of sequences of kernel functions in the computational device associated to the Controller. The Controllers can include policies to exploit concurrent kernel execution techniques, interleave computations with communications, or reorder the sequence of kernels. The kernel configuration is the selection of specific configuration parameters for the kernel launching, that can be associated to a particular kernel and computational platform.
- Data management, including the data transfers carried out across the memory hierarchies of the host and the accelerators, and the abstraction used to access data elements independently of the target device, the threads indexes space, or the data layout.

The Controller can be used as a layer to abstract the details of device management, thus generalizing the porting of programs across different type of accelerator devices. Several automatic code-generation compile-time tools [15, 42, 101] can derive from sequential code with pragma annotations: (1) The data dependences among the kernel launches; (2) The needed data transfers between the host and the target devices, and; (3) The domains on which kernels should be executed. Such tools have been used typically to generate code for only one kind of accelerator. For example, in [101], the OpenMP 4.0 *#pragma offload* feature is used to generate codes to execute on the XeonPhi accelerator. Using the same information extracted by this technologies, together with the abstractions and generic programming guidelines of our proposal, it would be possible to generate a generic code valid for any kind of execution device. This approach will be studied in a future work.

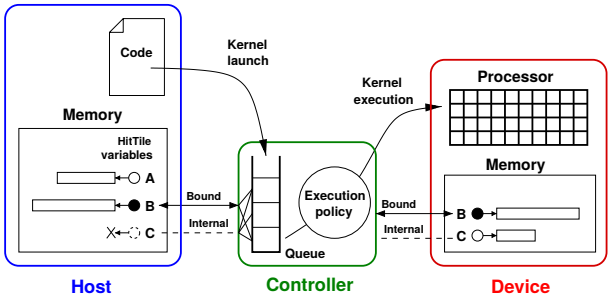

Figure 3.1: Diagram of the Controller model architecture. The kernel-launching requests can be enqueued. The Controller entity manages the execution of enqueued kernels, and the data transfers between memory spaces for bounded variables. In the figure, the host variable A is not bound to the Controller. Variable B is a bound variable, with a duplicated image for the data in the device memory. Variable C is an internal variable of the Controller, defined in the host, but allocated only in the device.

3.2.1 Kernel management

Kernel launching is an operation that inserts in the Controller queue an order to execute a kernel, with given real parameters, when the associated device (accelerator or set of CPU cores) is available (see Fig. 3.1). This is done using a method of the Controller. The Controller internals will ensure that the input data have been transferred to the device memory if needed, and that previous kernels have ended, before doing the real launch. The Controller execution policy could reorder the kernels in the queue to maximize the execution efficiency as far as the input/output dependencies across kernels are not violated. The main features and contributions of our model in the kernel management are described below:

Kernel definition and launching: Support of both generic and specialized codes

The model provides a generic mechanism to declare kernels.

We define a *kernel implementation* as a tuple: $(< name >, < deviceType >, < parameters\_list >, < code >)$, where *deviceType* is a symbol, or list of symbols (e.g. GPU, CPU), indicating the specific kind of device/s where the kernel can be deployed. Thus, we propose to allow the declaration of several implementations of the same kernel name, indicating the architecture/s for which this particular kernel is more suitable.

We also propose the possibility of defining kernel implementations for a *generic* device type. These kernels will be portable across different types of devices and architectures, and will be used by default when no specific kernel implementation is provided in the code for a given target device. As they are simply considered another kernel implementation, with a different device type symbol, they can be declared in the same program together with other implementations of the same kernel name. This allows the creation of libraries of kernel

implementations, developing first a generic and portable one that will work in any platform, and introducing gradually more specific and optimized kernel implementations for specific or new target devices.

The code of the generic kernels implementation, in order to be portable across different device types, should comply in our model with a set of properties (see also Sect. 3.3.6 for a discussion about an example of a generic kernel used on both multi-core CPU and GPU devices). Properties of a generic kernel: (1) The code is pure data-parallel, in the sense that it can be executed by many logical threads, and the programmer ensures that no data dependencies or race conditions can arise across them; (2) The code operates on indexed data structures using abstract 1, 2, or 3-dimensional thread indexes provided by the runtime system (t_x, t_y, t_z), that are internally adjusted for each device to access data in row-major order keeping maximum coalescing and/or vectorization properties in the specific device; (3) The code does not explicitly use resources, primitives, or synchronization mechanisms, that are specific of the programming model of a given device; (4) All accesses to data structures are done through an abstract interface provided by the runtime system, that is independent of the target device chosen (see Sect. 3.2.2 for our proposal to choose such an abstract interface).

A unified kernel launching function, at run-time, matches the architecture of the device to the best implementation provided at compile time for that architecture. It is possible to transparently use either generic portable kernels, optimized kernels programmed in the native models of a specific device, or even wrappers to call specialized libraries for specific device types. The Controller runtime system can choose the most appropriate one.

Finally, we propose to require in the parameters list of a kernel implementation, that the programmer reports the input/output *role* of each kernel parameter. This will be used by the Controller to detect which data should be transferred among different memory spaces, at each moment, depending on the sequence of kernels launched.

Characterisation of kernels for execution

Our model considers the characterization of the kernel code to automatically optimize launching parameters, such as the thread-block geometry. We propose to integrate the model of qualitative characteristics presented in [105, 131] in our Controller. To use this model, the programmers should examine the kernel code, and they should conceptually characterize it in classes according to three main criteria. We introduce an extension of the kernel implementation tuple with a new element to provide this classification of the kernel code. The Controller internally uses an associative table to implement the selection of launching parameters according to the rules proposed on the previously cited papers. More details about the criteria and examples are presented in Sect. 3.3.4.

3.2.2 Data management

Accelerators may have their own memory spaces, forcing to transfer the input data of the kernels between the memory of the host platform and the memory of the device accelerator,

and the obtained results back. The manual management of these data movements can be cumbersome and error-prone. Moreover, it is difficult to predict in advance when asynchronous data transfers are possible, or when data should stay in the accelerator device memory, as this depends on the exact sequence of kernels launching. Our model abstracts from the programmer all these issues. The main features/contributions of our model on data management are the following:

Data transfers between the host and the accelerators

In our model, a Controller is associated, in the moment of its creation, with a particular accelerator or subset of CPU cores, and it transparently manages in the accelerator memory space the images of the host memory data structures. The Controller can decide when and how the transfers should be carried out, depending on the data structures used in the corresponding kernels, and the sequence of kernels enqueued for launching. The programmer can use the original names of the host variables in the kernel transparently. Depending on the role of the variables (named data-structures) used as real parameters in kernel launches, we can distinguish two types: Bounded variables and internal variables.

Bound variables

Bounded variables are host variables that have an image in the memory space of the accelerator (see Fig. 3.1). The model defines an operation to bind a host variable to the Controller. Once a variable is bounded, its data should not be modified by the program at the host side until an unbinding operation is applied.

The first kernel requiring the use of a bounded variable as an input, will force the Controller to transparently ensure that its data have been already transferred before its execution. Applying an unbinding operation to a bounded variable will force the transfer of its data from the accelerator to the host, if it has been used as output by any kernel. The main program waits until the end of the kernels using that variable, and the end of the data transmission to the host if needed.

Internal variables

Internal variables are variables whose scopes are delimited to the kernels executed in the accelerator. They are only handled inside the memory space of the device, and they will not have allocation in the host memory space. Thus, they never imply a data transfer (see Fig. 3.1). In the particular case of a device representing a subset of CPU cores, the memory should be transparently allocated and managed in the host device.

The model defines an operation to create an internal variable in the Controller. A data structure is declared in the host without allocating memory to it. The name of this ghost data structure is used in the creation operation to clone the type, size, and internal structure in the Controller memory space and to keep track of the internal device pointer. Since that

moment, the name can be used as real parameter in kernel launching as a reference of the internal variable. To destroy an internal variable it is needed to apply another operation using the reference name of the host ghost variable.

Data accesses

One of the key features that a programming model for heterogeneous systems should provide is the ability to manage uniformly the data structures on a program. As previously commented when discussing generic kernel definitions, the data accesses on the body of the kernel should be independent of the target device chosen. We propose the use of abstract thread indexes and data-accessing methods in the kernel codes. They are devised in order to design the codes to work efficiently when accessing elements in row-major order, independently of the device. We propose the use of the data-handler abstraction for arrays introduced by Hitmap [55], a library for hierarchically distributed arrays. Efficient implementations have been provided for different devices, such as CPUs, or GPUs. See more details about the Hitmap functionalities used in the implementation of our model in Sect. 3.3.1.

3.3 The Controllers library

We have developed an implementation of the Controllers model. It is designed as a library written in C99 code. Thus, it can be used to develop C/C++ programs, independently of the chosen compiler. The library defines functions, but also relies on preprocessor macros for code rewriting. Although this allows an efficient interface implementation in a compiler agnostic way, the programmer should take care to use the interface in the expected way to avoid problems derived from some common pitfalls of macro substitutions (unexpected type checking issues, misnesting due to incorrect code injection in the macro parameters, etc.).

The current implementation supports NVIDIA's GPU devices using CUDA internally, and subsets of CPU codes using OpenMP internally. In our implementation, a kernel is a function coded for any, or for a specific computational device, with particular input/output parameters. The Controllers library interface defines primitives to: Create Controllers; Declare and characterize kernels; Manage host data structures that can be bounded or created as internal variables in the Controllers and transparently accessed inside the kernels; and Launch the kernels.

3.3.1 Data structures and Hitmap

Regarding the data structures that the model handles, we have decided to integrate our implementation with Hitmap [55], an efficient library for hierarchical tiling and mapping of arrays. It is based on an SPMD model, and on the message-passing paradigm. Hitmap has three main functionality modules: (a) Domain and tile management; (b) Mapping modules;

and (c) Communication patterns. Hitmap defines objects to declare and manipulate index sets as multidimensional parallelotopes with optional stride, or as sparse sets. Hitmap defines a plug-in system to include new mapping modules: Virtual topology constructors and mapping functions named *HitLayouts*. The modules generate objects that can be queried at runtime to obtain information about the result of the mapping for the local, or any remote process. Finally, it contains functionalities to build reusable communication patterns for tiles, or subtiles, across virtual processes. These functions internally use the MPI standard, exploiting efficient techniques like derived data types, and asynchronous communications.

Hitmap defines the *HitTile* structure, an abstract entity for n-dimensional arrays and tiles. A *HitTile* structure is a handler to store array meta-data, along with the pointer to the actual memory space of the data. The circles for the variables A, B, and C in Fig. 3.1 represent the handlers. The *HitTile* structure should be specialized for each array base type at the beginning of the program.

There are only four functions of Hitmap needed to work with the Controllers implementation. First, *hit_tileDomain* and *hit_tileDomainAlloc* are used to declare the index domains of a tile array, also allocating the memory for the data in the second case. The function *hit_tileFree* is used to free the data memory and clean the handler. The function *hit_tileElem* is used in host or kernels code to access the elements of a tile. It receives a tile name, a number of dimensions, and the indexes values of the desired element. This function provides an homogeneous interface to manage data structures in both, host and accelerator device kernels. The data are stored and accessed in row major order in all cases.

Hitmap library includes many other functionalities. They include management of hierarchical subselections of parts of the tiles, and transparent management of distributed-arrays, with abstract partition and communication functionalities that internally use a message-passing paradigm (exploiting MPI). This will allow in the future the transparent integration of the Controllers in distributed multi-node clusters.

Most of the meta-data in the *HitTile* structure are only needed for advanced functionalities, such as distributed-array partitions and communications in the host code. In our new implementation we define a new much smaller handler (*HitKTile*) with the minimum information needed for data accesses to multidimensional arrays through a data pointer. The kernel launching interface will transparently transform the handlers to this new type, substituting the data pointer with its equivalent in the device memory space. The data accesses inside the kernels uses this internal pointer along with a minimum number of arithmetic operations, exposing the expressions to the native compiler to open the possibility of further optimizations. The result obtains as good performance as the classical array accesses.

In Hitmap, the implementation of different kinds of tiles hides to the programmer the details of the data access for different internal data layouts. The Hitmap library already integrates sparse domains and sparse data structures into the HitTile abstraction [49]. The future transformation of these handler structures, to implement efficient and portable kernels, should follow a similar approach as the one used for dense arrays.

3.3.2 Controllers and variables management

Initialization and destruction

A Controller is associated to a particular computational platform (accelerator or CPU-cores set) at the moment of its creation. This is done through the `CtrlCreate` function (see line 13 of Fig. 3.3). This primitive has two main parameters: The name of the Controller variable, and the identification of the associated computing device. The programmer can free the resources with the `CtrlDestroy` function (see line 31 of Fig. 3.3).

The Controllers associated to CPU cores internally use OpenMP. To allow the launching of CPU kernels asynchronously, as in the case of GPUs, our implementation uses OpenMP tasks. A master thread executes the host code, and one OpenMP task is activated for each core associated to a CPU Controller. The Controller initializes on its creation an internal array of structures with one element per assigned core task, to control their activities and their synchronization.

Binding variables

The function `CtrlAttach` binds a tile defined in the code of the host with a Controller. The function `CtrlDetach` unbinds it (see lines 17 and 29 of Fig. 3.3 respectively). If the memory space of the device is not the same as the host, the attach operation allocates memory for the data in the device space. After the binding, the host should not manipulate the tile data until it is unbounded. During the time the variable is bounded, operations to copy the data to and from the associated device can happen at any time, as an internal decision of the Controller. For the particular case of Controllers associated to CPU cores, there is no data duplication. The kernels may be modifying the host variable data at any time directly.

In the current implementation we have integrated part of the variables management in the Hitmap tile handlers. The handler stores the pointer to the device memory, a reference to the Controller it is bounded to, and flags to indicate the clean/dirty state of the data. This avoids the duplication of bindings, attempts to unbind variables from the wrong Controller, etc. The flags help the Controller to choose the proper moments to synchronize the data between the host and the device memory image.

Creating internal variables

The function `CtrlInternalCreate` creates an internal variable on the device memory space. On the other hand, the function `CtrlInternalDestroy` is used to free the memory space in the assigned computational device (see lines 18 and 30 of Fig. 3.3 respectively). For the case of Controllers associated to accelerators, this kind of variables does not need to have allocated memory in the host side. A tile initialized with a domain information is enough. Even if it would have allocated memory, there is no synchronization of data between the host and the device image created by this functionality.

```
/* HitTile specialization
 * Creates new type HitTile_float */
hit_tileNewType( float );

/* Kernel characterizations */
KERNEL_CHAR(copyCell,1,def,def,def)
KERNEL_CHAR(updateCell,1,full,low,high)

/* Generic kernel codes for any device */
KERNEL(copyCell, 2, OUT, HitTile_float, dst, IN, HitTile_float, src){
  int x = thread.x; int y = thread.y;
  hit_tileElem(dst, 2, x, y) = hit_tileElem(src, 2, x, y);
}

KERNEL(updateCell, 2, OUT, HitTile_float, dst, IN, HitTile_float, src){
 int row = thread.x + 1;
 int col = thread.y + 1;
 hit_tileElem( dst, 2, row, col ) =
  (hit_tileElem(src, 2, row+1, col) + hit_tileElem(src, 2, row-1, col)
  + hit_tileElem(src, 2, row, col+1) + hit_tileElem(src, 2, row, col-1)
   ) / 4;
}
```

Figure 3.2: Examples of the kernel characterization and definition for a stencil program implementing an iterative Jacobi PDE solver for the Poisson's heat diffusion equation. The kernels are usable for both CPU and GPU Controllers.

```
/* Tile declaration and allocation (host) */
HitTile_float M, Mcopy;
hit_tileDomainAlloc(&M, float, 2, rows, columns);
hit_tileDomain(&Mcopy, float, 2, rows, columns);

/* Tile initialization */
for (int i=0; i<rows; i++)
  for (int j=0; j<rows; j++)
    hit_tileElem( M, 2, i, j ) = ...

/* Controller creation */
Controller ctrlGPU, ctrlCPU;
CtrlCreate(&ctrlGPU, COMM_GPU, 2);
CtrlCreate(&ctrlCPU, COMM_CPU, 4, 7);

/* Tile binding and creation (device) */
CtrlAttach(&ctrlGPU, &M);
CtrlInternalCreate(&ctrlGPU, &Mcopy);

/* Kernel launching */
CtrlThreads domain1 = CtrlThreads( 2, rows, columns );
CtrlThreads domain2 = CtrlThreads( 2, rows-2, columns-2 );
for (iter=0; iter< num_iterations; iter++) {
  CtrlLaunch(&ctrlGPU, copyCell, domain1, 2, Mcopy, M);
  CtrlLaunch(&ctrlGPU, updateCell, domain2, 2, M, Mcopy);
}

/* Unbinding and freeing resources */
CtrlDetach(&ctrlGPU, &M);
CtrlInternalDestroy(&ctrlGPU,&Mcopy);
CtrlDestroy(&ctrlGPU); CtrlDestroy(&ctrlCPU);
hit_tileFree(&M); hit_tileFree(MCopy);
```

Figure 3.3: Example of the main code, for a stencil program implementing an iterative Jacobi PDE solver for the Poisson's heat diffusion equation. The kernels are usable for both CPU and GPU Controllers.

3.3.3 Declaration and configuration of kernels

A kernel is declared by using the primitive `KERNEL_<type>` (see line 10 of Fig. 3.2). Where `type` may be empty to indicate a generic kernel, usable on any kind of device, or a specific value for a specialized code for a given type of device. This is useful when different optimizations on the kernel code are required for different devices. Currently, the library supports the specific primitives `KERNEL_GPU` for CUDA code targeting NVIDIA's GPUs, `KERNEL_CPU` for host machine code targeting sets of CPU cores, and `KERNEL_GPU_WRAPPER` for host machine code which includes calls to specialized GPU libraries, like for example cuBLAS routines. The Controller ensures that both kinds of GPU kernels are executed with exclusive control of the associated device. This allows the automatic coordination of the launching of sequences of classical kernels and calls to GPU libraries in the same device with our model.

The kernel-definition primitives declare in brackets the number of parameters of the kernel, with a tuple of information for each parameter (see again line 10 of Fig. 3.2). The parameter information includes its type, name, and input/output role:

- IN: for input HitTile parameters, whose elements are only read.
- OUT: for output HitTile parameters, whose elements are only written.
- IO: for input and output HitTile parameters, with elements can be both read and written.
- INVAL: for input parameters of any type passed by value.

For the case of accelerator devices, with separated memory spaces, this configuration allows the Controllers to determine if it is necessary to carry out data transfers with the main memory of the host when kernels are launched. The primitive is followed by a structured block with the kernel code. Thus, it resembles a C function header.

The Controller CPU implementation contains the loops to execute the kernel code for each element of the threads index space, that is internally assigned to a specific OpenMP thread. Both CPU and GPU kernels code use a predefined *thread* structure with three integer elements *x,y,z* indicating the 3-dimensional indexes of the corresponding fine-grain thread. In order to ease the kernel reuse across different device kinds, these indexes are adapted to have the same row-major meaning in both types of kernels. Also, the space of valid indexes that can be defined in the kernel launching is independent of the thread-block sizes in GPUs. Actual threads with identifiers outside the chosen launching index space, that could be added as padding to fill thread-blocks, will be clipped transparently by the Controller launching system. For specialized GPU kernels, the kernel code may use CUDA primitives mixed with the Hitmap data accesses that use the new portable index system.

3.3.4 Kernel characterization

The kernel characterization is a programmer hint to help the system to automatically determine proper kernel launching parameters in terms of special code features and platform architecture information. The CPU threads granularity in our prototype is determined by a simple regular blocking policy, that does not require a specific kernel characterization.

For GPU kernels, our current prototype library integrates the model presented in [104, 105, 131]. This model allows to determine configuration parameters (grid, thread-block and L1 cache memory sizes), for NVIDIA's GPUs. The primitive `KERNEL_CHAR`, taken from [106], is used to provide to the Controller the characterization of the kernels (see line 6 or 7 of Fig. 3.2). The primitive receives the kernel name, the number of dimensions of the thread space (1, 2, or 3), and descriptive values for the characterization model. These values are a qualitative description of characteristics of the kernel code provided by the programmer. They are related to: (a) The coalescing property of the global memory access patterns (full, medium, scatter); (b) The ratio of arithmetic/logic operations per global memory access (high, medium, low); and (c) The ratio of data sharing accesses in a block per global memory access (high, medium, low). For the default case, when the programmer cannot provide a proper characterization for all the parameters, a *def* keyword can be used instead of one of the given values, and the model provides typical thread-block values for each CUDA architecture, that work well in a general case, maximizing occupancy if possible, etc. For a more detailed description of this qualitative descriptors with tentative quantitative ranges, see the works [105, 106, 131].

Our implementation extends this characterization with the possibility to specify a fixed size for the thread-block. This is useful for kernels that rely on specific block sizes or geometries to manipulate shared memory (for example the matrix multiplication code in the CUDA Toolkit Samples).

3.3.5 Kernel launching

The function `CtrlLaunch` is used to launch a kernel, with given real parameters, to the computational device associated to a Controller (see line 24 or 25 of Fig. 3.3). The launched kernel will be enqueued, and eventually executed with the corresponding configuration derived from the information provided by the characterization primitives. Currently, the prototype only supports a First-Coming-First-Service policy for kernels execution. The launching function has the following parameters: (a) The Controller; (b) The name of the kernel; (c) The index space of the thread set; (d) The number of parameters required by the kernel; and (e) The real parameters for the kernel execution.

The dimensions of the threads index space are specified using a *CtrlThreads* structure that stores up to three integer values representing the cardinality on each dimension. It is equivalent to the *dim3* type in CUDA, but it is used for both, GPU or CPU kernels independently. The variable parameters should be tile variables associated to the Controller, internal or bounded, or any value of the proper type for the INVAL parameters.

```
/* CUDA kernel launch */
/* Parameters:
  name: Kernel name
  threads: CtrlThreads, index space limits
  arch: GPU architecture
  params: real parameters of the launch
  kchar: characterization of the kernel
*/
dim3 block = CALBlockModel(name,kchar,arch);
dim3 grid = CALGridDivUp(threads,block);
wrapper_gpu_##name<<<grid, block>>> (threads, params);
...
__global__ void wrapper_gpu_##name( ... ) {
   CALThread threadId = { threads.dims,
     threadIdx.y + blockDim.y * blockIdx.y,
     threadIdx.x + blockDim.x * blockIdx.x,
     threadIdx.z + blockDim.z * blockIdx.z };

   if ( threadId.x >= threads.x  || threadId.y >= threads.y ||
        threadId.z >= threads.z ) return;

   kernel_gpu_##name( threadId, params );
}
```

Figure 3.4: Excerpt of the Controller library code generated for kernel deployment/launching on a CUDA capable GPU device. The block geometry is selected using the kernel characterization provided by the programmer. We also show the part of the wrapper function launched, that reverses the thread indexes before calling the actual kernel code.

```
/* OpenMP index space partition and kernel launch */
/* Parameters:
  name: Kernel name
  n_th: OpenMP thread identifier
  numCores: number of CPU-cores associated to the current controller
  threads: CtrlThreads structure with limits of the threads index space
  params: real parameters of the kernel launch
*/
int stripSize =(int) ceil( threads->x/(numCores) );
int begin= n_th*(stripSize);
int end= MIN(first + (stripSize) - 1,(threads->x)-1);
for(i=begin; i<=end; i++)
   for(j=0; j<threads->y; j++){
         threadId.x = i;
         threadId.y = j;
         threadId.z = 0;
         kernel_cpu_##name(threadId, params);
     }
```

Figure 3.5: Excerpts of the Controller library code generated for kernel deployment/launching on a group of CPU-cores. Each OpenMP thread (assigned to a core) executes this code. For simplicity, in this example, the 2-dimensional index space is divided into blocks by rows without balancing the remaining. The code executes the loops that call the kernel code for each index-element mapped to the thread. It simulates the many-thread approach used for GPU programming using coarser-grain OpenMP tasks.

Internally, the execution of a GPU kernel implies: (1) The creation of the small HitKTile handlers using the original tile handlers information; (2) The use of the characterization model to select the grid and thread-block geometries; and (3) The execution of a wrapper function. This wrapper reorders the thread indexes to be used as in the CPU kernels code to access data elements in row-major order, and ensures that threads outside of the required index space return immediately before executing the kernel code (see an excerpt of this code on the Fig. 3.4). For CPUs, the kernel execution internally implies the partition of the index space in coarse blocks. Figure 3.5 shows an example of a simplified code that performs this partition. The kernel launching (line 17) calls to an inlined function that is generated when this kernel implementation for CPU-cores is declared. A final global synchronization is needed to preserve the launching semantic before proceeding to launch another kernel.

3.3.6 Programming example

In this section we present a practical application of the library concepts described in the previous section by using an example. Figure 3.2 and 3.3 shows the implementation of a Stencil 2-D application. On Fig. 3.2, first, we declare at the beginning of the program the specialized Hitmap array types to be used in the program (see line 3 in Fig. 3.2).

Kernel characterization: Two kernels are characterized at lines 6 and 7 of Fig. 3.2. The first kernel characterization uses the default configuration, while the second one looks for a particular configuration for the *updateCell* kernel. This last characterization is based on the code properties. According to the classifications presented in [105, 131], the global accesses of the program suggest an almost fully coalesced memory access pattern. The ratio of arithmetic operations per data element is low because less arithmetic operations are done compared to read/write operations. The ratio of data shared between threads of the same block is high, because most of the neighbor values are reused across their computations.

Kernel definition: The kernel definitions start in lines 10 and 15. The kernel definitions include the name of the kernel, the number of parameters, a specification for each parameter of its input/output mode and type, and the parameter name. Both are generic kernel codes valid for any kind of device as the *KERNEL* primitive points out. The code in the kernel bodies is executed in parallel on the device by as many threads as we choose in the main program. Another kernel definition example can be seen on Fig. 3.6 (right). In this case, it is a specific kernel code for GPUs that calls a function of a specialized library for NVIDIA's GPUs.

Data management: All the accesses to the variables in the kernel bodies (either reads or writes) must be done through the Hitmap functions. For example, line 12 of Fig. 3.2 shows the copy of an element of the matrix *src* on the matrix *dst*. It uses the *hit_tileElem* function to access the data element corresponding to the indexes assigned to the logical thread that executes the kernel. We can see another example on lines 18 to 21 of Fig. 3.2. The *dst* matrix element corresponding to the thread indexes is updated using the values of the neighbor cells in the matrix *src*.

```
/* Recurrence equation kernel */
KERNEL_CHAR(kRecurrence, 2,
     full, high, low)

/* Black Scholes kernel */
KERNEL_CHAR(kBlackScholes, 1,
     full, medium, low)

/* Stencil Jacobi kernels */
KERNEL_CHAR(kCopy, 2,
     full, low, low)
KERNEL_CHAR(kUpdate, 2,
     medium, medium, medium)

/* GPU matrix mult. kernel */
KERNEL_CHAR(kMatrixMult, 2,
     fixed-square-32)
```

```
/* Kernel wrapper for cuBLAS
 * matrix mult. */
KERNEL_GPU_WRAPPER(HitTile_float A,
                   HitTile_float B,
                   HitTile_float C){
 const float alpha = 1.0f;
 const float beta  = 0.0f;
 cublasHandle_t handle;

 cublasSgemm(handle,
   CUBLAS_OP_N, CUBLAS_OP_N,
   B.dimx, A.dimx, A.dimy, &alpha,
   hit_ktileRawData(B), B.dimx,
   hit_ktileRawData(A), A.dimx,
   &beta, hit_ktileRawData(C),
   B.dimx );
}
```

Figure 3.6: Characterization of the generic or GPU specialized kernels for the case studies (left). Example of kernel wrapper to execute a specialized GPU library function (right).

We show on Fig. 3.3 the code of the main function that coordinates the application execution.

Data structures: Data structures are created and initialized in the first part of the main program (see lines 1 to 9 in Fig. 3.3). The matrix *M* is allocated on the host, where it is initialized. Thus, we use the *hit_tileDomainAlloc* function to create the data structure. However, the *Mcopy* matrix is an internal variable, and does not need to have allocation in the host memory space. Only a virtual representation is created, without assigning actual memory, by using the *hit_tileDomain* function.

Controller entity: Lines 13 and 14 of Fig. 3.3 show examples of the Controller creation. One associated to the third GPU of the system, and another one associated to the subset of CPU cores with indexes in the range 4 to 7. Once the Controller is created, host data structures are binded to the target devices, and internal data structures are created in the target devices through the Controller (see lines 17 and 18 of Fig. 3.3).

Kernel launching: With the data structures already associated to the target device, we can launch the kernels. The kernel would be executed by as many threads as a *CtrlThread* object specifies. We see in lines 21 and 22 how two *CtrlThread* objects are created. The first one includes the domain of the whole data structure, and the second one the whole data structure without the borders. After that, in lines 24 and 25 the two kernels are launched using their different thread-index domains, by using the different *CtrlThread* objects. After the desired number of iterations of the kernels, the control of the result data structure is

transferred again to the host by unbinding it, and the Controllers are destroyed (see lines 29 to 31). The CPU Controller is not used in this example code.

To summarize, we observe that the final program presents an organized sequence of programming phases, that leads to a clear structure. Easy generic guidelines to program with this library can be deduced from this simple example.

3.4 Experimental study

This section describes the experiments we have carried out to check the functionality, and to evaluate the potential performance issues, introduced in the Controller prototype implementation. We also evaluate the development effort of using the Controllers prototype when compared to directly using common native programming models (CUDA or OpenMP).

3.4.1 Case studies

As case studies we have selected the following programs. All of them work with floating point numbers. From the CUDA Toolkit Samples, we have selected the *Black-Scholes* program, and two *Matrix Multiplication* examples, one using GPU shared-memory, and another directly calling the optimized *CUBlas* routine. We also use the stencil program that is shown in Fig. 3.3. Finally, we have selected a code to independently apply a trivial recurrence equation to elements in a matrix. A quick description and the motivation to choose these examples follows. The characterizations of the kernels using the proposed model, are shown in the left of Fig. 3.6. The parameters where chosen following the guidance presented in [105, 131].

Recurrence equation

This code uses two input matrices A, B, and writes a result matrix C that is originally initialized with zeros. The kernel computes the first 500 terms of a trivial recurrence equation that involves only two single addition operations: $C_x(i,j) = C_{x-1}(i,j) + A(i,j) + B(i,j)$, providing as a result the last term computed for each position. There is only one kernel launching for the whole program. It is an embarrassing parallel code, where each element is computed independently. The code involves a high number of independent and repetitive coalesced reads and writes to global memory on each thread. This artificial code is specifically designed to test the efficiency of the data accesses through the new small tile handlers and the generic data-access interface inside the kernel codes. A common kernel is used for both CPU and GPU Controllers.

Black-Scholes

The Black-Scholes formula is based on a mathematical model of a financial market. The result estimates the price of European-style options. The program in the CUDA Toolkit Sample independently applies the formula to a chosen number of input values stored in an array, calculating and storing their results. Thus, it is also an embarrassing parallel program with perfectly coalesced accesses on a GPU. Each thread does only one read and one write operation to global memory. It applies several floating point operations calculating intermediate result stored in registers or temporal variables. The kernel is called 512 times consecutively. This program is adequate to measure the cost of multiple kernel calls of more sophisticated arithmetic computation, with a very low number of global memory accesses per thread. A common kernel is used for both CPU and GPU Controllers.

Stencil computation

This program computes the stability point of a Partial Differential Equation (PDE), in this case the Poisson's equation for heat diffusion. It uses a Jacobi iterative method on a 2-dimensional discretized space, represented as a matrix. The program implementation is shown in Fig. 3.3. It is an 4-point stencil program that executes a fixed number of time iterations. On each iteration it independently computes a new value for each cell in the matrix, using the information of the four neighbor cells.

This kernel presents a similar number of arithmetic operations per thread as global memory accesses. The read operations are not completely coalesced due to the neighbor accesses, and the positions read by each thread are overlapped with the neighbor threads. A neighbor synchronization is needed at the end of each iteration, before a new simulation step starts. This kernel measures the effects of non-completely coalesced global memory accesses.

These stencil simulation programs are usually optimized to swap the two data structures (the one read, and the one write) after each iteration. We have intentionally skipped this optimization, implementing another kernel that simply copies the new generated data from the written variable to the original one. It shows the ability of the characterization model to provide different launching parameters for different kernels in the same program. The program is based on a repetitive invocation of two kernels with very low load per thread. Thus, it tests the efficiency of the implementation of the kernel launching procedures. Again, the same kernel is defined for both CPU and GPU platforms.

Matrix multiplication

The Matrix multiplication computes the product of two different square matrices, storing the result in a third one: $C = A * B$. The computation of each cell of the resulting matrix is not dependent on another computation. Nevertheless, different cells use elements of A or B that are also read by other cell computations. Thus, data can be reused and shared across the computation of each cell. Moreover the read patterns on A and B matrices should be studied

to exploit coalescence in GPUs, and properly exploit caches in CPUs. These lead to different interesting optimizations in both GPU and CPU devices. Thus, we use different specialized kernels for each kind of device.

A direct simple solution to this problem involves one generic kernel, using a bidimensional grid of threads, for both CPU and GPU. Each thread $t_{i,j}$ is responsible of computing the dot product operation ($\sum_{k=0}^{n-1} A[i][k] * B[k][j]$), storing the result in the (i, j) position of the C matrix. Nevertheless, the GPU implementation in the CUDA Toolkit Samples exploits the shared-memory for better performance. The threads on each thread-block can use shared-memory to collectively load a square block of A and B matrices in a coalesced way. Then, they can efficiently perform a block matrix multiplication using the elements on the shared-memory. Several iterative stages should be applied to compute all the matrix block multiplications needed at each block of threads. Threads need to use block synchronizations on the global memory read operations, using specific CUDA code. This code, due to the way it uses the shared memory and it aligns the read operations, forces the use of a specific square thread-block size (32×32). It is an example of the utility of the extension to the characterization model introduced in our implementation.

For the GPU Controllers we have a second implementation. It shows the functionality of our kernel-wrapper facility, that allows to implement and launch as a kernel a call to the optimized cuBLAS routine for matrix multiplication (see the right of Fig. 3.6). The wrapper kernels do not need characterization, as they contain only host code, and the library routines include the code that take the decisions about launching parameters.

The current CPU kernel version is the generic simple implementation of the dot product of a row of A and a column of B to compute the result for a single output element. Further optimizations based on loop reordering and tiling for better cache usage can be automatically be applied by the native C compiler.

3.4.2 Development effort and code complexity

The first part of our experimental study evaluates how the use of our proposed model affects the development effort when compared with using the native programming models for the two types of devices considered in the current version of the prototype: CUDA for GPUs, and OpenMP for multi-core CPUs.

We measure three classical development effort and code complexity metrics: COCOMO lines of code, number of tokens, and McCabe's cyclomatic complexity. The first two ones measure the volume of code that the programmer should develop. The third one measures the rational effort needed to program it in terms of code divergences and potential casuistry that should be considered to develop, test, and debug. The metrics are applied to the part of the code that includes the kernel, the functions invoked by them, and the host coordination code. We ignore input data initialization, error or results checking, performance instrumentation, and writing messages to the standard output. Thus, the considered host code includes the declaration, creation and initialization of Controller, data containers, and structures

Case study	Version	Lines of Code	#Tokens	Cyclomatic Complexity
Recurrence equation	CUDA	44	404	5
	Ctrl.GPU	33	315	3
	OpenMP	27	243	4
	Ctrl.CPU	36	339	3
Black Scholes	CUDA	148	903	8
	Ctrl.GPU	106	704	7
	OpenMP	81	539	7
	Ctrl.CPU	90	707	6
Jacobi solver	CUDA	43	445	8
	Ctrl.GPU	40	371	4
	OpenMP	33	310	6
	Ctrl.CPU	47	456	5
Matrix mult.	CUDA	71	614	5
	Ctrl.GPU	47	429	4
	OpenMP	22	235	4
	Ctrl.CPU	40	389	4

Table 3.1: Measurements of the development effort metrics for the codes of the case studies.

for parallelism coordination, the memory transfer between host and devices in the native implementations, and the kernel launching operations in both cases.

Table 3.1 shows the measurements for these metrics in the baseline versions, using OpenMP for the CPU variant and CUDA for the GPU variant, and the versions using our Controllers library interface. The results indicate that programming with Controllers for GPUs generates a significantly lower volume of code, and a reduced cyclomatic complexity, indicating a clearly lower development effort than using CUDA. A closer look at the codes indicates that most of the reduction is found in the host part of the codes, as expected. On the other hand, for this kind of data parallel codes, the comparisons with OpenMP codes show that using the Controllers interface reduces a little the cyclomatic complexity, but increases the code volume.

The main advantage of considering the CPU cores as an accelerator device in the Controller model is found in the portability of code between GPUs and CPUs. Table 3.2 shows the percentage of words that can be reused, should be deleted, or should be changed to port from a CUDA program to the equivalent OpenMP version; and the same measurements when porting from a Controller version for GPUs to the equivalent Controller version for CPUs. The results clearly indicate that the portability of the Controller versions across different device types is really high in the three first cases, and still significant for the matrix multiplication program, that includes different specialized kernels. In this case the coordination

Case study	CUDA → OpenMP	Ctrl.GPUs → Ctrl.CPUs
Recurrence equation	Common 14%	Common 95%
	Delete 29%	Delete 4%
	Change 57%	Change 1%
Black-Scholes	Common 56%	Common 72%
	Delete 32%	Delete 17%
	Change 11%	Change 11%
Jacobi solver	Common 13%	Common 91%
	Delete 42%	Delete 0%
	Change 44%	Change 9%
Matrix multiplication	Common 8%	Common 49%
	Delete 4%	Delete 3%
	Change 88%	Change 48%

Table 3.2: Comparison in terms of the percentage of words that are common and can be reused, should be deleted, or should be changed, when porting codes between GPU and CPU versions using the native models, or the Controllers model.

code using Controllers is still the same, deriving in a significantly lesser number of words to change than when porting between CUDA and OpenMP.

3.4.3 Performance study

The second part of our experimental study measures the impact of using our Controllers prototype in terms of performance. We have run the GPU implementations in an NVIDIA's GeForce Titan Black X, with CUDA capability 5.2, installed on a host machine named *Hydra*, with two CPUs Intel Xeon E5-2609 v3 @1.90GHz, and 64Gb DDR3 main memory. To test the CPU implementation we have used a shared-memory machine with a higher number of cores. It is named *Heracles*, and it is a Dell PowerEdge R815 server, with 2 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores per processor, and a total of 32 cores. The operating system in both machines is a Linux Centos 7 OS. The programs have been compiled using the CUDA Toolkit 7.5, and GCC 4.8.3. We have used the flags *-arch=compute_52*, *-O3*, and the *-mtune* choice appropriate for each host machine.

We present comparisons of the execution times skipping input data initialization, results checking, and control messages writing. The instrumented code includes Controllers creation and variables binding/unbinding operations (which may imply data transfers). In the same way, CUDA data copies between the host and the device memories are also included. In the case of GPU programs we also measure the execution time accumulated by the kernels launching alone, without memory transfer operations. Due to instabilities on the execution

Case study		Recurrence	Black-Scholes	Jacobi	Matrix Mult.
Number of iterations		500	512	200	-
CPU Data size		$15\,000 \times 15\,000$	$1\,000\,000$	$10\,000 \times 10\,000$	$3\,000 \times 3\,000$
GPU Data Size	S	$10\,000 \times 10\,000$	$1\,000\,000$	$1\,000 \times 1\,000$	$3\,008 \times 3\,008$
	M	$15\,000 \times 15\,000$	$10\,000\,000$	$5\,000 \times 5\,000$	$5\,024 \times 5\,024$
	L	$20\,000 \times 20\,000$	$50\,000\,000$	$10\,000 \times 10\,000$	$10\,016 \times 10\,016$
	XL	$25\,000 \times 25\,000$	$100\,000\,000$	$20\,000 \times 20\,000$	$15\,008 \times 15\,008$

Table 3.3: Input data sizes and number of iterations selected for each case study in the performance experimental study.

		Recurrence		BlackScholes		Jacobi		Matrix Mult.	
Size	Measure	CUDA	Ctrl.GPU	CUDA	Ctrl.GPU	CUDA	Ctrl.GPU	CUDA	Ctrl.GPU
	Host	0.2580	0.2559	0.0084	0.0047	0.0033	0.0023	0.0357	0.0274
S	Kernels	0.5592	0.5723	0.0409	0.0454	0.0151	0.0167	0.0663	0.0666
	Total	0.8172	0.8283	0.0493	0.0500	0.0184	0.0191	0.1020	0.0941
	Host	0.5720	0.5721	0.0617	0.0654	0.0672	0.0676	0.0495	0.0416
M	Kernels	1.2753	1.2752	0.3862	0.3951	0.3203	0.3285	0.2878	0.2983
	Total	1.8473	1.8469	0.4479	0.4604	0.3875	0.3960	0.3373	0.3399
	Host	1.0249	1.0236	0.2955	0.2899	0.1167	0.1168	0.4977	0.4944
L	Kernels	2.2502	2.2502	1.9166	1.9221	1.2277	1.2306	2.1012	2.0925
	Total	3.2750	3.2739	2.2121	2.2120	1.3444	1.3474	2.5989	2.5869
	Host	1.5909	1.5864	0.5954	0.5695	1.1901	1.1915	0.4191	0.5748
XL	Kernels	3.4668	3.4669	3.8311	3.8652	4.8493	4.8574	7.0198	7.0203
	Total	5.0577	5.0529	4.4265	4.4346	6.0394	6.0490	7.4389	7.5441

Table 3.4: Execution time (seconds) for the case studies versions using CUDA, or Controllers for GPUs, with different input sizes. Host time measures the data communications, coordination code, and launching operations. Kernels time measures the actual kernel execution times. Total times are the addition of both times for easy comparison.

times of the host codes and data transfers, the results always show the mean of the execution time obtained on 5 repetitions of each test.

For the CPU versions we have selected input sizes with enough load to achieve scalability in our test machine with up to 32 threads. The exact input size parameters chosen are presented in the first row of Table 3.3. The total execution times of the baseline programs and the versions based on Controllers are presented in Fig. 3.7, for different number of active threads/cores. The results indicate that our implementation of the Controller abstraction for CPUs does not implies significant overheads for the scalability ranges tested.

For a fair comparison, both GPU approaches (CUDA and Controller based) use the same values for the configuration of launching parameters. In the case of the Black-Scholes program, the thread-block size value predicted by the characterization model is the same

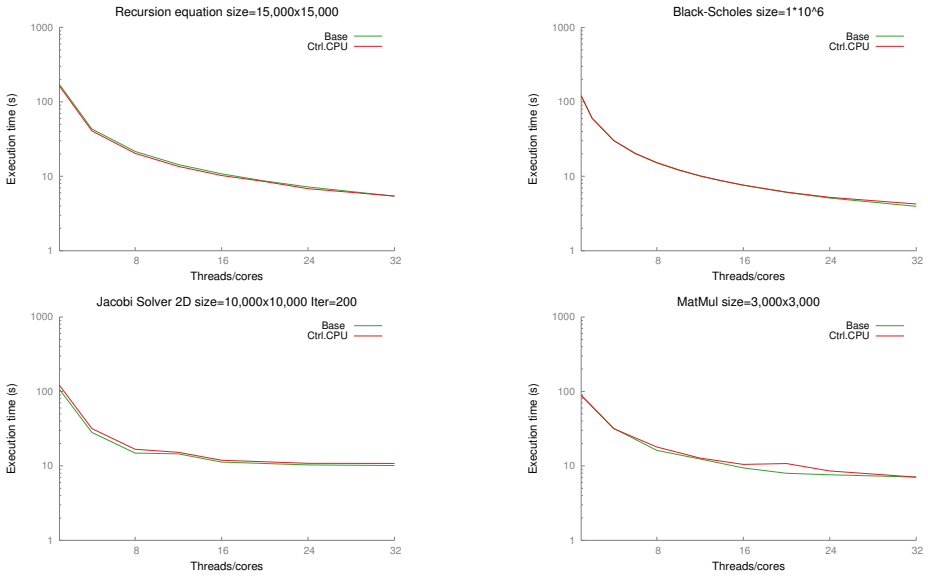

Figure 3.7: Execution times (seconds) in Heracles machine of the baseline (Base) and the Controller versions for CPU device (Ctrl.CPU) with a variable number of cores.

found in the original CUDA code. In the matrix multiplication case, it is fixed in both cases to 32×32. For the other two case studies, the values predicted by the characterization model are injected manually in the baseline versions. These values have shown to produce at least 96% of the best performance obtained with any other thread-block geometries. We compare the performance obtained when launching the baseline and the Controllers versions with different input data sizes, selected to produce from very low to significant execution times (from tenths of a second, to more than four seconds). The exact input size parameters are presented in Table 3.3, with a class name (S, M, L, XL) for easier reference in the following discussion.

The results for GPU programs are presented in Table 3.4. We skip the presentation of times for the matrix multiplication using the cuBLAS library. The memory transfers and launching operations are enclosed into the library routine, that is called by both the CUDA and the wrapper virtual kernel in the Controller version. Thus, the performance is the same except for stochastic behaviors that affect both codes equally.

The execution times of the kernels show a little overhead when using the Controller interface, due to an extra internal stage for thread index conversion and threads space clipping. Nevertheless, the overhead is always less than 1% of the kernels mean execution time. The recurrence equation kernel, specifically designed to test the efficiency of the data accesses through the small kernel tile handlers, shows the minimum overhead of the four cases.

As expected, the execution times of the host codes, in both CUDA and Controller versions, are smaller than the corresponding kernels execution, specially for large input sizes (L and XL), that present significant kernel execution times. These times spent on host code coordination, and memory transfers, are less stable than the kernels execution times. Sometimes, the Controller versions show even better results than the equivalent host code in the CUDA versions. Although, the same memory transfers in the CUDA codes are internally executed in the Controller versions, memory transfers have the less predictable times, with a higher variance. The rest of the Controller operations are in general light loaded. The critical one is the kernel launching, that involves several checks and handler manipulations. Nevertheless, the overhead of the launching operation has been measured to be less than 1×10^{-5} seconds for the GPU implementation, in the machine used for this study. Thus, the accumulated overhead of many launching operations in the Black-Scholes, or the Jacobi programs, is not significant even for the smaller input sizes tested (class S). This overhead could only be clearly noticeable when executing long sequences of kernels with really low computational load, which are not in general appropriate for parallelism exploitation, specially in an accelerator.

3.5 Summary

In this chapter we proposed the Controller model, a parallel programming model that simplifies the coding of applications for systems compound by computational units of different kinds. It is based on an abstract entity, the Controller, that abstracts to the programmer the launching of kernel sequences on GPU accelerators, or sets of CPU cores.

It provides mechanisms to: (1) Associate Controllers to devices; (2) Define portable kernels that can be reused across different types of devices; (3) Define specialized kernels for the same program on different device types; (4) Automatically select proper values for launching parameters on different devices through a characterization of the kernels provided by the programmer, and; (5) Automatically deal with different memory spaces of the host and the devices when needed. This model unifies the kernel programming and data structures management, bringing closer the accelerator and multi-threaded programming, and taking into account the architectural differences of the accelerator platforms to obtain good performance. Our experimental study shows the advantages of using this approach in terms of development effort metrics, and the efficiency of our prototype implementation for several case studies representing different computational costs, and global memory accesses scenarios.

Chapter 4

Supporting the Xeon Phi coprocessor in the Controller Programming Model

Different kinds and families of accelerators, such as GPUs, CPUs or Xeon Phis, are used in modern high-performance platforms, as we observe in the configuration of the TOP500 supercomputers [130]. This chapter addresses the second goal of the Thesis:

Support for new kind of accelerators such as the Xeon Phi Co-processor.

In this chapter, we extend the Controller heterogeneous programming model presented in the previous chapter, in order to include support for Intel Xeon Phi coprocessors, also known as Many Integrated Cores (MICs). The model is based on the mix of the communication model originally designed for GPUs in the Controller library with the execution model originally designed for groups of CPU cores. We develop a complete runtime execution system that includes methods for task launching, transparent data transfers between the MIC accelerator and the host, and a queue system to manage the kernel executions with a customized grain choice. It perfectly fits with the previous Controller library, thus standardizing and abstracting to the programmer the issues related to the programming of different kinds of accelerators. It provides a MIC runtime support for a heterogeneous programming model, that unifies the programming for heterogeneous systems composed by MIC coprocessors, GPUS, or CPUs, also obtaining the same performance than using their native programming models.

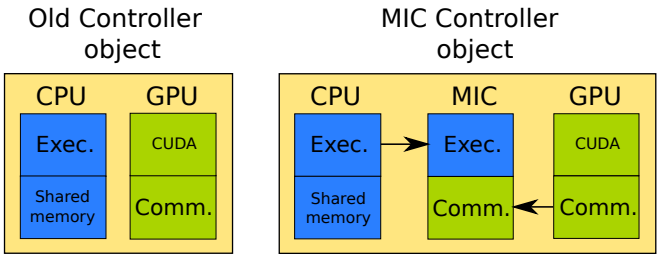

Figure 4.1: Left: Previous Controller model, only supporting CPU and GPU. Right: The MIC Controller proposed model, mixing features from the GPU-CPU submodels.

4.1 Approach to support MIC accelerators

For this proposal, we distinguish two internal parts in the previous version of the Controller, which provide support for each kind of computational device: Execution and communication management models (see left of Fig. 4.1). The abstract device formed by CPU-cores share the memory space with the host. Thus, the controller object only has to provide an execution model that manages a task queue and that adapts the fine-grain computations used in the Controller model to a coarser granularity, more appropriate for CPU threads. On the other hand, for GPUs, the CUDA programming model already provides an execution system to enqueue and launch kernels with the same granularity level used in the Controller model. However, the communication operations across different memory spaces on host and GPUs require the implementation of a more sophisticated mechanism to integrate different policies and techniques in the Controller model.

In this new proposal, we decouple both the execution and the communication management models for different kind of devices, that are independent parts in the Controller abstraction. Thus, we propose their mixture to support a new type of accelerator such as the MIC coprocessors. In order to do that, we use: (1) In terms of execution, the Controller model for groups of CPU cores, that blends blocks of fine grained kernels into coarse CPU tasks, is appropriated for MIC coprocessors. (2) In terms of memory management, the abstract model for data communications needed for the MIC coprocessors is equivalent to the GPU communication model in the old Controller approach.

The application of this idea leads to a homogeneous programming model for heterogeneous systems including MIC coprocessors, where the issues related to the programming of different types of accelerators are transparent for the programmer. In this work we show the implementation of this idea in the Controller library, to support computational devices such as MIC coprocessors, GPUs, and groups of CPU-cores, without redesigning or changing the high-level programming model and interface.

```
/* Matrix addition: Generic kernel code for any type of device */
KERNEL(MatAdd, 3, OUT, HitTile_float, C,
 IN, HitTile_float, A, IN, HitTile_float, B ){
    int x = thread.x; int y = thread.y;
    hit_tileElem( C, 2, x, y ) = hit_tileElem( A, 2, x, y ) +
                                 hit_tileElem( B, 2, x, y );
}
/* Host program using the Controller library */
int main(){
    int SIZE = 10000;
    /* Stage 1: Controller creation */
    Cntrl comm;
    CntrlCreate(&comm, CNTRL_GPU, 0);
    /* Stage 2: Data structures creation and initialization */
    HitTile_float A; HitTile_float B; HitTile_float C;
    hit_tileDomainAlloc( &A, float, 2, SIZE, SIZE );
    hit_tileDomainAlloc( &B, float, 2, SIZE, SIZE );
    hit_tileDomainAlloc( &C, float, 2, SIZE, SIZE );
    initMatrices(&A, &B, &C);
    /* Stage 3: Data structures attachment */
    CntrlAttach(&comm, &A); CntrlAttach(&comm, &B);
    CntrlAttach(&comm, &C);
    /* Stage 4: Kernel launching */
    CtrlThreads threadsSpace = CtrlThreads( 2, SIZE, SIZE );
    CntrlLaunch(comm, MatAdd, threadsSpace, 3, &A, &B, &C);
    /* Stage 5: Data structures detachment */
    CntrlDetach(&comm, &C);
}
```

Figure 4.2: Kernel definition and configuration, and host program of a matrix addition using the Controller library.

```
/* Internal attach function */
void attachToXPHI(CntrlXPHI* ctrl,
                  HitTile *tile){
Lock(tile, ctrl);
int MIC= ctrl->MIC;
char *data = (char*)(*tile).data;
int numElems = hit_tileSize(tile);
#pragma offload target(mic:MIC) \
      in( data:length(numElems) \
      alloc_if(1) free_if(0) )
}
```

```
/* Internal detach function */
void detachToXPHI(CntrlXPHI* ctrl,
                  HitTile *tile){
int MIC= ctrl->MIC;
char *data = (char *)(*tile).data;
int numElems = hit_tileSize(tile);
#pragma offload target(mic:MIC) \
   in( data:length(0)           \
        alloc_if(0) free_if(0)) \
    out( data:length(numElems)  \
        alloc_if(0) free_if(1))
 Unlock(tile, ctrl);
 }
```

Figure 4.3: Excerpts of the internal codes that perform data transfers of a HitTile object. Left: From the host to a MIC coprocessor. Right: From a MIC coprocessor to the host.

4.2 Integrating MIC coprocessors in the Controller library

The original version of the Controller library supports the deployment of kernels on GPUs or virtual computational devices formed by groups of CPU-cores. In this section, we describe a method to integrate the support of MIC coprocessors in this model. We implement the **MIC controller object** containing several functionalities, such as the identification, initialization and management of MIC devices, an adapted internal queue to manage the asynchronous kernel execution mechanism to transfer data from/to the co-processor memory, and a method to lock accesses to the HitTile data structures on the host while they are managed in the device memory.

Figure 4.2 shows a matrix addition implementation that performs the computation on a GPU using the Controller model. The proposal of this chapter allows the efficient execution of this program on the Xeon Phi, only by changing the `CNTRL_GPU` parameter by a new `CNTRL_XPHI` parameter on the line 13 of the code of Fig. 4.2.

4.2.1 Attaching and detaching data structures on the MIC

In computational devices such as GPUs or MIC coprocessors, whose memory spaces are separated from the host memory space, the attachment/detachment operation also implies a data transfer.

We have implemented two internal functions to perform the data transfers to/from the MIC coprocessor, using the Intel Language Extensions for Offload (LEO). These functions are executed internally when the program invokes an attachment or a detachment operation respectively. Figure 4.3 shows a summarized version of the code of both functions.

On the left, we see the code used to attach a tile to a MIC controller object (represented in the figure by the `CntrlXPHI` type). In this function, first the attached tile is locked on the host. Second, the code extracts: 1) The MIC identifier assigned to the controller object (line 5); 2) The pointer to the actual data (line 6); and 3) The number of bytes to be transferred (line 7); After that, the function performs the actual data transfer from the host to the MIC, ensuring that there is allocated memory space in the target device (using `alloc_if(1)`), and that after this offloading the actual data will be maintained (using `free_if(0)`).

On the right, we show the code used to detach a tile whose data has been modified from a MIC controller object. As in the attachment, first the code extracts the information about the data transfer (lines 4 to 6). Second, the actual data transfer from the coprocessor to the host is specified using a pragma. For determining the pointer of the data previously transferred, the program uses the `in` modifier to make the data pointer available in the Xeon Phi, and sets the *length* to 0 to prevent any data from being copied (lines 8 to 9). Once the pointer is available on the MIC, the pragma also specifies the data transfer and the freeing of the MIC space memory (lines 10 to 11). Finally, the data structure is unlocked on the host.

4.2.2 New kernel definitions

A kernel definition specifies the device that fits with the contained code by declaring it using the primitive `KERNEL_<type>`. We extent the Controller framework to support also MIC kernel definitions. A MIC kernel definition is rewritten as three functions using macro functions. We show examples of the code of the three resulting functions in Fig. 4.4.

Fine-grain virtual thread function: The first function implements the kernel code that the programmer defines to be executed for one index element of the fine-grain virtual threads space. In most array operations, it is used to compute one data element. In Fig. 4.4, lines 4 to 6 show the function declaration and lines 39 to 41 the function definition. The function is named `kernel_xphi_##name`, where *##name* is the kernel name, taken from the first parameter of the kernel definition primitive. It is defined as a MIC function using the attribute `target(mic)`. The parameters are a multi-dimensional index, that represent a point in the execution thread-index space, represented by a `CtrlThread` object, and the actual kernel parameters.

Parallel coarse-grained function: The second one (`wrapper_xphi_##name` starting in line 9 of Fig. 4.4) performs the offloaded coarse-grained parallel computation in the MIC device. It receives a variable number of parameters. The first one is the controller object, the second one the domain of fine-grain thread indexes to compute and the rest are the data structures corresponding to the real parameters. Lines 11 to 13 of Fig. 4.4 show how the information is extracted from the parameters (auxiliary macros for the transformations were defined in Fig. 4.5). The rest of the body of the function defines the offload region. The offload pragma transfers the data-structure handlers, the selected index space of logical threads represented by a `CtrlThreads` object, and the pointer to the actual data for each HitTile. As in the detachment operation, in order to determine the data pointers previously

```
/* Macro of the kernel definition */
#define KERNEL_XPHI(name, nparams, params...)                          \
  /* Single-element function declaration */                            \
  static void __attribute__((target(mic)))                             \
      kernel_xphi_##name(CtrlThreads threadId,                         \
       XPHI_WRAPPER_PARAMS##nparams(params));                          \
                                                                       \
  /* Parallel coarse-grained function */                               \
  static inline void wrapper_xphi_##name(void** args){                 \
    int MIC=cntrl->MIC;                                                \
    CntrlXPHI* cntrl = (CntrlXPHI*) args[0];                           \
    CtrlThreads* threads = (CtrlThreads*)args[1];                      \
    XPHI_WRAPPER_CAST##nparams(params);                                \
    _Pragma( STRINGIFY(XPHI_OFFLOAD_PARAMS##nparams(MIC, params)))     \
    {                                                                  \
     XPHI_POINTERS##nparams(params);                                   \
    _Pragma("omp parallel"){                                           \
     int i,j,k;                                                        \
     CtrlThreads threadId;                                             \
       _Pragma("omp for private(i,j,k)")                               \
       for(i=0; i<=threads->x; i++){                                   \
         for(j=0; j<=threads->y; j++){                                 \
           for(k=0; k<=threads->z; k++){                               \
             threadId.x = i;                                           \
             threadId.y = j;                                           \
             threadId.z = k;                                           \
             kernel_xphi_##name(threadId,                              \
                    XPHI_WRAPPER_VALUES##nparams(params));             \
       } } }                                                           \
     }}                                                                \
                                                                       \
  /* Task addition function */                                         \
  void name##_xphi(CntrlXPHI* cntrl, CtrlThreads thread,               \
                   XPHI_WRAPPER_PARAMS##nparams(params)){              \
      CntrlXPHIAddTask(cntrl, wrapper_xphi_##name, thread, nparams,    \
                       XPHI_WRAPPER_VALUES##nparams(params));          \
  }                                                                    \
  /* Single-element function definition */                            \
  static void __attribute__((target(mic)))                             \
      kernel_xphi_##name(CtrlThreads threadId,                         \
       XPHI_WRAPPER_PARAMS##nparams(params))                           \
```

Figure 4.4: Functions internally generated by the MIC kernel definition: 1) Function to be executed by each fine-grain virtual thread: `kernel_xphi_##name`; 2) Function that executes a dequeued kernel, grouping virtual threads in coarse-grained OpenMP threads: `wrapper_xphi_##name`; 3) Function to enqueue a kernel-launching request: `name##_xphi`.

```
1  /* Auxiliar macros for kernels with one parameter */
2  #define STRINGIFY(a) #a
3  #define XPHI_WRAPPER_PARAMS1(io1, type1, value1)                    \
4          type1 value1
5  #define XPHI_WRAPPER_VALUES1(io1, type1, value1)                    \
6          value1
7  #define XPHI_WRAPPER_CAST1(io1, type1, value1)                      \
8          type1 value1_p = (type1)args[2];                            \
9          HitTile value1_t = *(HitTile*)value1_p;                     \
10         char *data_tile1= (char *) (value1_t).data;
11 #define XPHI_OFFLOAD_PARAMS1(MIC, io1, type1, value1)               \
12         offload target(mic:MIC) in(threads:length(3)) in(value1_t)  \
13                   in(data_tile1:length(0) alloc_if(0) free_if(0))
14 #define XPHI_POINTERS1(io1, type1, value1)                          \
15         HitTile value1 = value1_t;                                  \
16         value1.data = data_tile1;
```

Figure 4.5: Auxiliary macros defined for a one parameter kernel.

transferred, the offload pragma uses the `in` modifier to make the data pointer available in the Xeon Phi, and sets the *length* to 0 to prevent any data from being copied (see line 14 of Fig. 4.5). Inside the offload region, the HitTile handlers update their data pointer to the actual offloaded data (line 16). After that, the parallel computation is performed on the specified domain (lines 17 to 30), grouping virtual thread indexes in actual coarse-grained threads, by using an OpenMP parallel loop.

Kernel launch request: The third function is named `name##_xphi`. See lines 33 to 37 of Fig. 4.4. It is the internal Controller implementation of a kernel launch for a MIC. In its body, the function implements the enqueuing of the kernel execution request in the Controller object. The information needed is: The controller object, the pointer to the coarse-grained parallel computation function, and its real parameters (the index space where the application will be executed, the number of kernel parameters, and the actual kernel parameters).

4.2.3 Queue management and Kernel launching

As opposite to the CUDA programming model, the offloading MIC coprocessor programming model does not provide a queue system to manage asynchronous kernel launchings. We have developed a queue system for the asynchronous execution of several kernel launches on the MIC coprocessor, currently using a FIFO policy in our prototype. When a MIC controller object is created, an asynchronous `OpenMP task` is launched. This task uses OpenMP locks to block until there are kernel-launching requests in the queue. Then, it dequeues the request and dispatches/executes it. The execution of a task on the MIC is carried out by simply executing

the already offloaded parallel `wrapper_xphi_##name` generated function, specified in the request structure, that contains pointers to the function and parameters. The Controller destructor enqueues a special request that notifies to the OpenMP queue-controlling task that it should release the Controller resources and finish.

4.3 Experimental study

We perform an experimental study to evaluate the potential advantages and constraints of the integration of the MIC coprocessor in the original Controller library. The section includes: (1) A description of the considered study cases, (2) A performance study of our proposal, and (3) A development effort comparison between programming using the new Controller extension and using device vendor programming models.

4.3.1 Study cases

We select four benchmarks to test our approach and implementation.

Matrix addition. It implements a sum of two matrices, storing the result in a third one: $C = A + B$. For the Controller version, we use the same generic kernel implementation tested in previous works for CPU-cores and GPUs, without any modification.

Black-Scholes. As in the Sect 3.4.1 of Chap. 3, we choose the Black-Scholes program, obtained from the CUDA Toolkit Samples. Again, the Controller version uses the same generic kernel definition for both GPUs, and MICs accelerators.

Matrix multiplication. It computes the product of two matrices, storing the result in a third one: $C = A * B$. The read patterns on A and B matrices should be adapted to exploit coalescence and shared memory in GPUs, and to properly exploit caches and vectorization on MICs. These features lead to different optimizations in both types of accelerators. Thus, the Controller version declares different specialized and optimized kernels for each kind of device.

Mandelbrot algorithm The Mandelbrot algorithm is used to compute fractal geometric images. The *escape time algorithm* is the simplest algorithm for generating a representation of fractal geometric images of the Mandelbrot set. Its implementations does not need any data transfer to start the computation, only to return the results, presenting also an irregular workload per thread. The Controller version uses a single generic kernel definition for both GPUs and MICs accelerators.

4.3.2 Performance study

In this section we show how low is the performance overhead produced by the implementation of our proposed MIC library extension. Table 4.1 shows the total times spent (including computation and data transfers) by the four benchmarks with two different problem sizes.

Code	Mat. Add.	Black-Scholes	Mat. Mult.	Mandelbrot
Size	5000^2	10^6	4096^2	4000^2
LEO Code	1.67	0.60	2.59	6.49
Ctrl. Code	1.43	0.74	2.88	6.86

Code	Mat. Add.	Black-Scholes	Mat. Mult.	Mandelbrot
Size	20000^2	$5 * 10^7$	8192^2	20000^2
LEO Code	24.99	5.49	19.87	148.47
Ctrl. Code	24.65	5.01	19.27	147.36

Table 4.1: Performance results (seconds) comparing LEO reference codes with Controller codes for different input sizes. Experiments executed on a Intel Xeon E5-2620 v2 @2.1GHz, 32Gb DDR3 main memory, and with the Xeon Phi Knights Corner 3120A coprocessor. Compiler used: ICC 17.0.0 version with the flags *-O3*, and *-openmp*.

Codes have been implemented with our proposal, and directly with the Intel Language Extensions for Offload (LEO) and OpenMP. This study indicates only a small constant penalty performance due to the management of the queue system, that is only noticeable in the results for the smaller problem sizes presented on the left of Tab. 4.1. For bigger problem sizes, some performance gain is obtained due to Hitmap optimizations in the internal management of the data structures. In general terms, the performance obtained by using our approach is similar to the native programming models.

4.3.3 Development effort measures

This section includes two development effort comparisons. First, between the proposed Controller implementation and the reference codes (using LEO and OpenMP for MIC). Secondly, comparing measures of the code changes needed to port a GPU implementation to a MIC implementation, using the Controller or the native programming models.

The results of the first comparison are presented on Tab. 4.2. We measure three classical development effort metrics: Number of lines of code; Number of tokens, and McCabe's cyclomatic complexity [86]. The measured codes include kernel definitions, kernel characterizations, the coordination host code, and data structures management. We observe that the use of the Controller library implies less cyclomatic complexity, but more number of lines and tokens.

However, the goal of the library is to provide an homogeneous interface to deal with any kind of accelerator. For this reason, we also compare the effort needed for transforming GPU codes in order to port them to a MIC device. See results on Tab. 4.3. We analyze the percentage of words of each implementation that are common and can be reused, should be deleted, or should be changed. The largest changes are on the matrix multiplication benchmark, because

Case study	Version	Lines of Code	#Tokens	Cyclomatic Complexity
Matrix addition	LEO	26	210	3
	Ctrl.MIC	35	317	1
Black Scholes	LEO	80	525	6
	Ctrl.MIC	89	693	5
Matrix mult.	LEO	23	217	4
	Ctrl.MIC	37	337	3
Mandelbrot	LEO	32	319	5
	Ctrl.MIC	46	488	4

Table 4.2: Comparison of number of code lines, code tokens, and cyclomatic complexity between the Controller version and the version using native programming models.

Case study	CUDA → LEO	Ctrl.GPUs → Ctrl.MICs
Matrix Addition	Common 13%	Common 92%
	Delete 30%	Delete 0%
	Change 57%	Change 8%
Black-Scholes	Common 53%	Common 69%
	Delete 25%	Delete 21%
	Change 22%	Change 10%
Matrix multiplication	Common 8%	Common 49%
	Delete 43%	Delete 3%
	Change 48%	Change 47%
Mandelbrot	Common 32%	Common 97%
	Delete 61%	Delete 0%
	Change 7%	Change 3%

Table 4.3: Comparison in terms of the percentage of words that are common and can be reused, should be deleted, or should be changed, when porting codes between GPU and MIC versions using the native models, or the Controller library.

of the implementation of different optimized kernels for each device but with our proposal they are still less than for the native versions. For the other benchmarks, we see that using our proposal the programming effort needed to change the target computational device is extremely low (it could be null in some cases parametrizing the controller type using an execution argument). These measures show the level of abstraction and standardization achieved by our proposal.

4.4 Summary

In this chapter we proposed an extension to the Controller programming model and to its library implementation, in order to support the Intel Xeon Phi (MIC) coprocessors. To provide support for MIC coprocessors, our approach reuses and mixes the internal execution features for CPU-cores, and the internal memory and communication management features of the original GPU model. We have completely integrated the support for a MIC coprocessor in the Controller library, without adding any constraint to the programming model. The experimental study shows the high flexibility of our approach, that implies a minimum programming effort for changing the execution target devices, without significatively penalizing the performance. Future work includes the integration of scientific libraries, such as MKL, as kernels in the Controller implementation, and an evaluation with applications of other domains.

Chapter 5

Multi-Device Controllers: A library to simplify the parallel programming of multiple heterogeneous devices.

Current HPC clusters are composed by several machines with different computation capabilities and different kinds and families of accelerators. Programming efficiently for these heterogeneous systems has become an important challenge.

In this chapter we address the third goal:

Design and develop an adaptable, simple, and efficient programming model for heterogeneous systems.

In order to do that, we introduce the *Multi-Controler (MCtrl)*, an abstract entity that coordinates the management of heterogeneous devices, including accelerators with different capabilities and sets of CPU-cores.

5.1 Introduction

Different number and types of accelerators or CPU cores may be available in the same machine. Exploiting all these devices together to solve the same problem has become an important challenge, due to the different computational units (CPUs, GPUs, XeonPhi, ..), that appear in a cluster machine, typically have different programming requirements and constraints to achieve the best performance [121].

There are many proposals to simplify the programming and management of accelerator devices, and the hybrid programming mixing accelerators and CPU cores, as we state in Sect. 2.3. However, in many cases, portability compromises the efficiency on different devices. Depending on the proposal, some details concerning the coordination of different types of devices are still tackled by the programmer, such as computation partition and balance, data mapping and locality, or data movement coordination across different memory hierarchies.

In this chapter we introduce the *Multi-Controler (MCtrl)*, an abstract entity implemented in a library, that coordinates the management of heterogeneous devices, including accelerators with different capabilities and sets of CPU-cores. It helps the programmer to handle the computation partition, mapping, and transparent execution of complex tasks in such hybrid and heterogeneous environment, independently of the target devices selected at run-time. Our proposal allows the exploitation of simple generic kernels that can fit in any device; very specialized kernels defined and optimized by the programmer for each architecture; and even wrappers to call third-party predefined libraries (such as e.g. cuBLAS [103]). This allows the exploitation of native or vendor specific programming models, in a highly efficient way. The most appropriate kernels for each target device are automatically selected by the entity during program execution. Our proposal improves state-of-the-art solutions, simplifying the data partition, mapping, and transparent deployment of both, simple generic kernels portable across different device types, and specialized implementations defined and optimized using specific native or vendor programming models (such as CUDA for NVIDIA's GPUs, or OpenMP for CPU-cores).

Our work is developed on the concept of *Controller* presented in the previous chapters. While the Controller transparently manages the data movements and the launching of series of kernels on a given target device, the proposed Multi-Controller coordinates several Controllers associated to different devices or groups of CPU-cores. It can be implemented as an extensible library, using the best programming models, tools, and compilers for each potential device.

We present an experimental study with five case studies. We show that our approach is highly flexible, with minimum programming effort for changing the target devices. The results of a performance study, comparing our approach with optimized reference codes show that our implementation does not introduce significant performance penalties.

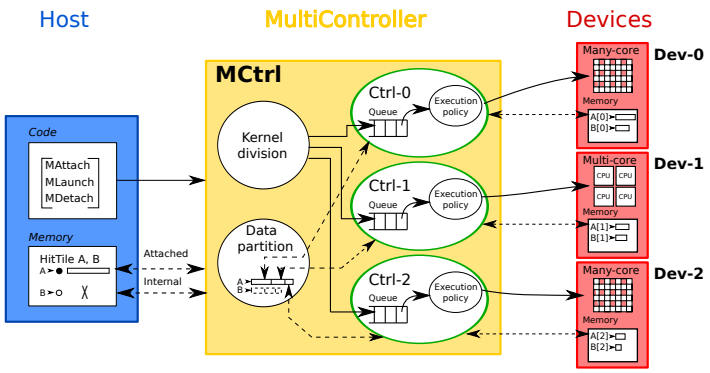

Figure 5.1: Diagram of the Multiple-Device Controller library (MCtrl).

5.2 Multiple-Device Controller (MCtrl) library

The Multiple-Device Controller (MCtrl) library provides a simplified way to program applications targeting heterogeneous systems with different kinds of computational units in the same machine. We define computational unit/target device as an accelerators (GPU, Xeon Phi, etc.) or a group of CPU-cores considered as a single independent device. The goal of this library is: (1) to automatize the data partition and data transfer between the host and multiple target devices, as well as (2) to transparently coordinate the division and execution of the computation among different computational units, independently of the kind of target device exploited (GPUs, group of CPU-cores, etc).

The library has an object-oriented design, despite the fact that it is mainly developed in C language. The classes are implemented as C structures with associated functions. The Multi-Controller model architecture is presented in Fig. 5.1.

The Multi-Controller object provides functions to manage:

- **Multi-device coordination**: The Multi-Controller is associated with a set of different devices at construction. It internally creates Controller objects to interact with each device. The Multi-Controller provides an abstract interface that enables it to manage it as a single computational device, independently of the internally associated devices.

- **Data structures**: The Multi-Controller abstraction creates an unified memory context for all of the associated devices, where internal data structures can be created, or data structures from the main host thread can be attached. Data structures can be replicated, or partitioned and distributed across the devices, depending of the program requirements. In the current prototype a simple static partitioning method has been included. It divides the structures into as many irregular parts as number of devices were selected in the Multi-Controller construction. The size of each part is calculated

proportionally from a list of weights. Data movements across different device memory hierarchies are transparently managed by the internal Controller objects associated to each device.

- **Kernel definition and launching**: The Multi-Controller model integrates the Controllers idea of multi-version kernel definition. Thus, kernel launching in a Multi-Controller simply uses a kernel name. The internal Controller selects, at run-time, the most appropriated kernel version or implementation for the associated device, form those provided by the programmer. The Multi-Controller internally divides the computation associated to a kernel launching among the different devices. The kernel execution on a device is performed asynchronously with respect to the kernels execution in the rest of devices. Synchronizations are required only by data requests on the main host thread.

5.2.1 Multi-Controller construction

A Multi-Controller object is constructed to manage a specific collection of devices. Its construction functionality receives an ordered list of device specifications. Each device specification is used to internally create a Controller object associated to the computational resource. In the current prototype we support device specifications that include: (1) NVIDIA's GPUs, specifying their CUDA device number, (2) Groups of main host CPU-cores, specifying a range of core identifiers, according to their internal numbering in the CPU information provided by the operating system, and (3) Xeon Phi co-processors, indicating its identifier number.

The Multi-Controller internally creates a queue to temporarily store the requests for kernel launchings, before dividing the computation and mapping it to the queues of the internal device Controllers. The synchronization and coordination operations of each Controller are executed on its own task, which makes the use of a host thread asynchronous, only when activity is needed, for minimal interference with other host threads. In the current prototype, the internal device Controllers are implemented using OpenMP tasks.

5.2.2 Data structures and domains

One of the objectives of the Multi-controller library is to provide an homogeneous interface to work with data structures in different device types, preserving the coalescing or vectorization properties of the code due to the data accesses order. The previous Controllers library uses Hitmap to provide such an interface (recall Sect. 3.3.1).

For Multi-Controllers, we propose to exploit and extent Hitmap functionalities to provide a transparent abstraction for the partition, subselection, and mapping of parts of the data structures to the different devices associated to a Multi-Controller. The Multi-Controller model proposes a single memory context for the whole set of associated heterogeneous devices. Data structures from the main host thread can be attached to the Multi-Controller

context, and they should not be manipulated on the main host thread until they are detached from the Multi-Controller. The Multi-Controller can decide when the real data movements should be done, synchronously or asynchronously, to the actual devices, depending on the kernels enqueued for execution, and their data dependencies.

To avoid redundant data movements across memory hierarchies, the model provides the programmer with a flexible attachment functionality. The current proposal is focused on applications where: (1) No data transfers between devices are needed across several kernel executions; and (2) any part of the computation needs, either a whole data structure, or a subset of the data structure that does not overlap with other subparts. Thus, data structures should be assigned as a whole to all the devices, or partitioned in non-overlapping parts, one for each device.

To support this model, we have extended the HitTile objects in the Hitmap library with the capability of dividing itself into several sub-selections, and store the information about the partition inside the object. Internally, when a HitTile is attached to a Multi-Controller, it performs the following steps:

1. First, it checks that the HitTile is not already attached to any Multi-Controller. If that particular HitTile object is already attached, the program raises an error, as a second attachment could lead to race conditions due to the concurrent execution of kernels in different Multi-Controllers.

2. If it is an attachment without the partition option, the whole space of indexes of the data structure are mapped to each device. If it is an attachment with the partition option activated, the Multi-Controller divides the index space of the HitTile data structure in a number of parts equal to the number of devices defined in the Multi-Controller. The partition policies introduced in Hitmap are responsible for dividing the data structure with no-overlapped domains. With the basic heterogeneous-oriented partition policy module included in our prototype, the partition size corresponding to each device is proportional to the weights provided in an array of floating point numbers, one for each device. More sophisticated partition policies can be easily added in the future thanks to the modular plug-ins system in the Hitmap library. The information about the mapping is stored in the meta-data of the HitTile object for further reference.

3. Finally, the Multi-Controller creates a HitTile structure for the space or sub-space of indexes mapped to each device, and proceeds to carry out the data transfers to the assigned device when needed. Transfers are not needed for groups of CPU-cores of the host, or accelerators that can shared the host memory space. The transfer policies inside the Multi-Controller can take decisions about when and how to make the transfers. For example, the current Multi-Controller prototype implement both, immediate and lazy transfers. The implementation of asynchronous transfers is currently an on-going work.

When the data structure is detached, the Multi-Controller object ensures the consistency of the whole data structure in the main host thread. This may imply data transfers from some or all of the associated devices. The information stored in the objects about the index space mapped to each device is used for the transfers, and eliminated at the end of the detachment procedure. The semantic of this operation makes it synchronous. The main host thread should block until its state is consolidated.

Inherited from the Controllers library, the Multi-Controller model also allows the attachment of HitTile structures that have a defined index space, but no memory allocated in the main host thread. This creates partitioned internal memory buffers (replicated or partitioned) inside the space of the devices, which are transparently treated inside the kernel functions as any other data structure. The detachment of these structures simply frees the corresponding subparts and internal resources in the devices.

In the case of partitioned data structures, the actual parameters used in the execution invocations of the Multi-Controller are substituted by the the HitTiles created by subselecting the mapped portion of the index spaces for each device. Inside the kernel functions they are transparently used as normal whole HitTile data-structures. The kernel launching interface will transparently transform the HitTile handlers for the real parameters handlers to an internal HitKTile type, substituting the data pointer with its equivalent in the device memory space when needed. The data access primitives used inside the kernels code are transparently rewritten to use the pointer contained in these objects, along with a minimum number of arithmetic operations, to access the data. The resulting code exposes the arithmetic expressions to the native compiler to open the possibility of further optimizations. The result obtains as good performance as direct array accesses in static codes.

5.2.3 Kernel launching

The Multi-Controller model proposes a unified space of indexes for logical threads across the whole set of associated heterogeneous devices. One instance of the kernel function is executed by each logical thread. This model directly fits with the *threads grid* abstraction in current GPUs programming models such as CUDA or OpenCL. In the case of groups of CPU-cores, the internal Controller objects are responsible for executing the kernel invocations of a grid of many logical threads inside a limited set of coarse threads (e.g. one OpenMP thread per core) for efficiency (See Sect. 3.3.5).

The Multi-Controller kernel launching function receives as parameters, the name of the kernel, the real parameters for the kernel (whole data structures attached to the Multi-controller, or single typed values), and a definition of the indexes space for the logical threads.

Internally, the Multi-Controller performs, at runtime, the intersection between the indexes subset defined by the grid, or domain of threads, specified by the programmer, and the data structures domains or sub-selections performed by the attachment partition procedure. The result points to the domains of logical threads where each device should perform the computation. Figure 5.2 shows a graphical representation of an example of this

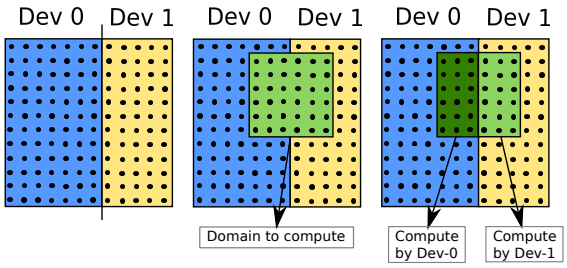

Figure 5.2: Calculating the domains to compute for each device.

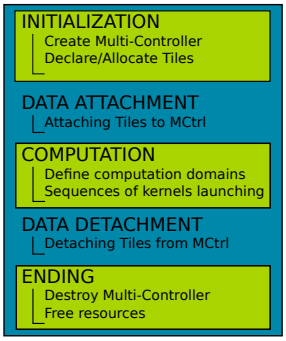

Figure 5.3: Typical programming stages using the MCtrl library. The data-structures attachment/detachment, and kernel launching stages can be repeated and interleaved as desired.

procedure. Stage 1 of the figure shows an example of the partition of a data-structure domain. Stage 2 overlaps the domain (grid of threads) where computation is required. Stage 3 shows the domains of logical threads that should be executed on each device. With this method, the computation is transparently divided as a function of the data partitions previously performed.

With the information about how the computation is divided, the Multi-Controller object deploys the kernel launches on the internal Controller objects with a non-empty sub-space of threads mapped. Thus, the global computation launching is subdivided into sub-kernels that are enqueued for execution in their corresponding device queues.

The information provided by the programmer in the characterization primitives of the kernel declarations is internally used to determine launching parameters for the appropriate devices, such as thread-block geometries for GPUs.

5.2.4 Programming methodology and example

In this section we discuss, using an example, how a program is developed using our proposal. The proposed methodology derives in clearly structured programs, using simple development guidelines. Figure 5.3 shows the typical stages of an application programmed using the Multi-Controller library. After the creation of the Multi-Controller object, the data structures declared in the main host thread can be attached to the Multi-Controller object. Computations are started defining the threads space and invocating kernel launchings in the Multi-Controller object. The detachments consolidate the state of the whole data structures, modified during the computation, in the main host thread.

Figure 5.4 presents two codes of a matrix addition programmed using our proposal. On the left of Fig. 5.4, a group of ten CPU-cores and a GPU are exploited, assigning to them the 10% and the 90% of the computation to each, respectively. On the other hand, in the code on the right, two GPUs are exploited for computation. In this case, each GPU performs 50% of the computation. Both codes follow the basic structure of programming stages presented in Fig. 5.3.

First, the programmer creates a Multi-Controller object. The creation of this controller includes the definition of the different kinds of devices controlled by this object, as well as a parameter specifying their computation features. The parameter, when the device is a group of CPU-cores, corresponds to the range of CPU-cores used. For a GPU device, this parameter indicates the GPU identifier (see lines 7 to 9).

The second step consists of the data structures creation and allocation. HitTile objects are created and allocated using a HitShape object that represent a domain. To do that, we have applied the function `hit_tileDomainAlloc(...)` (see lines 16 to 24). HitTiles are attached to the Multi-Controller using the `MCtrlAttach(..)` function (see lines 29 to 34). For the attachment, using the current partition policy in the prototype, it is needed an array of floats indicating the weights used to divide the data structure among the different target devices. The Multi-Controller is responsible for the actual data attachment on the final devices where the computation will be performed.

After that, the computation domain (indexes domain for logical threads) is defined by a CtrlThreads object (see lines 38 and 39). Typically, the kernel execution is performed for each element of a matrix, as in all the cases studied in this chapter (see Sect. 5.3.1). In this cases, the computation domain and the data structure domain are equal.

The kernel launching is performed in lines 41 to 47. The parameters of the `MCtrlLaunch` function are: (a) The Multi-Controller object; (c) The domain that defines the computation space index; (c) The name of the kernel; (d) The number of parameters required by the kernel; and (e) The real parameters for the kernel execution. It deploys the kernel executions on all the computational devices associated to the Multi-Controller. It internally enqueues, in each device Controller, a copy of the kernel launching, adapting the thread index domain with the goal of each target device will perform only its corresponding part of computation.

Finally, once the computation has finished, HitTiles are detached from the controller, the Multi-controller is destroyed, and the data structures are freed (see lines 49 to 61).

```
// MCtrl creation
CALMCtrl cntrlMult;
// Two devices:
//   a 10 CPU-core group,
//   and a GPU
CAL_MCtrlCreate2(cntrlMult,
          CAL_CNTRL_CPU,
                  RANGE(0,9),
          CAL_CNTRL_GPU, 0 );

// Specifying the weights
// corresponding to each device
float percents[2] = {10, 90};

// Define whole data structures
HitTile_float A, B, C;
HitShape domain;
domain=hit_shapeStd2(SIZE, SIZE);
hit_tileDomainAlloc(A, 2, float,
                    domain);
hit_tileDomainAlloc(B, 2, float,
                    domain);
hit_tileDomainAlloc(C, 2, float,
                    domain);

// Attach the data structures to
// a MDC, determining the weights
// for each device
CAL_MCtrlAttach(A, cntrlMult,
                percents);
CAL_MCtrlAttach(B, cntrlMult,
                percents);
CAL_MCtrlAttach(C, cntrlMult,
                percents);

// Determine the threads
// to launch.
CtrlThreads threads;
threads=CtrlThreads(2,SIZE, SIZE);

// Perform the commputation
CAL_MCtrlLaunch(cntrlMult,
                threads,
                MatAdd, 3,
                hit_CM(&C),
                hit_CM(&A),
                hit_CM(&B) );

// Copy result from MDC memory
// to host memory
CAL_MCtrlDetach(A, cntrlMult);
CAL_MCtrlDetach(B, cntrlMult);
CAL_MCtrlDetach(C, cntrlMult);

// Destroy MCtrl
CAL_MCtrlDestroy2(cntrlMult);

//Free CHitTiles
hit_Free(A);
hit_Free(B);
hit_Free(C);
```

```
// MCtrl creation
CALMCtrl cntrlMult;
// Two devices: two GPUs


CAL_MCtrlCreate2(cntrlMult,
          CAL_CNTRL_GPU, 0,

          CAL_CNTRL_GPU, 1 );

// Specifying the weights
// corresponding to each device
float percents[2] = {50, 50};

// Define whole data structures
HitTile_float A, B, C;
HitShape domain;
domain=hit_shapeStd2(SIZE, SIZE);
hit_tileDomainAlloc(A, 2, float,
                    domain);
hit_tileDomainAlloc(B, 2, float,
                    domain);
hit_tileDomainAlloc(C, 2, float,
                    domain);

// Attach the data structures to
// a MDC, determining the weights
// for each device
CAL_MCtrlAttach(A, cntrlMult,
                percents);
CAL_MCtrlAttach(B, cntrlMult,
                percents);
CAL_MCtrlAttach(C, cntrlMult,
                percents);

// Determine the threads
// to launch.
CtrlThreads threads;
threads=CtrlThreads(2,SIZE, SIZE);

// Perform the commputation
CAL_MCtrlLaunch(cntrlMult,
                threads,
                MatAdd, 3,
                hit_CM(&C),
                hit_CM(&A),
                hit_CM(&B) );

// Copy result from MDC memory
// to host memory
CAL_MCtrlDetach(A, cntrlMult);
CAL_MCtrlDetach(B, cntrlMult);
CAL_MCtrlDetach(C, cntrlMult);

// Destroy MCtrl
CAL_MCtrlDestroy2(cntrlMult);

//Free CHitTiles
hit_Free(A);
hit_Free(B);
hit_Free(C);
```

Figure 5.4: Matrix addition example programmed using our approach: Exploiting a group of 10 CPU-cores and a GPU for the computation (left); and exploiting two GPUs for the computation (right).

5.3 Experimental study

This section presents an experimental study to show how this approach simplifies the programmer effort to adapt programs to different sets of heterogeneous devices, and the efficiency obtained by our prototype implementation of the Multi-Controller library. First, we present several benchmarks used in the study, discussing their features. Second, we present some development effort measures. Finally, we provide a comparison of performance measures obtained by programs using device vendor or native programming models, and programs developed with the Multi-Controller library.

5.3.1 Study cases

We have used four common benchmarks as base-lines for five case studies to test our proposal.

Matrix addition

The Matrix addition consists of the sum of two different matrices, storing the result in a third one: $C = A + B$. The computation of each cell does not imply any kind of dependencies with the computation of another one. The solution developed using our proposal involves just one kernel with a bidimensional grid and bidimensional thread-blocks. Depending on the size of the grid and the matrices, each block of threads computes the resulting values of several blocks of the matrix iteratively, following the example implementation presented to the CUDA programming guide [102]. The accesses to global memory are fully coalesced. This benchmark requires a large quantity of data transfers to the accelerators used, while the computation load is really low. Using our model, the CPU solution for this problem is similar to the GPU version. Only one generic kernel should be defined by the programmer.

Mandelbrot algorithm

We use the same example used in Sect. 4.3.1. The irregular load of each thread in this example, because of the input coordinates chosen, is evenly distributed across the image rows. Thus, the computation space can be divided between two GPUs giving the same amount of work to each GPU, producing good results. This example will clearly show the balancing effect of partitioning the workload among several GPUs. The application does not need any data transfer to start the computation, only to return the results. The parameters chosen for the experiments are: An image size of 2048×2048, and a limit of $60,000$ iterations per pixel. A visual representation of the area chosen is shown in Fig. 5.5. The GPU thread-block size used is 32×32 for all the experiments with this benchmark. In our implementation, the same generic kernel definition is used for both GPUs and CPUs.

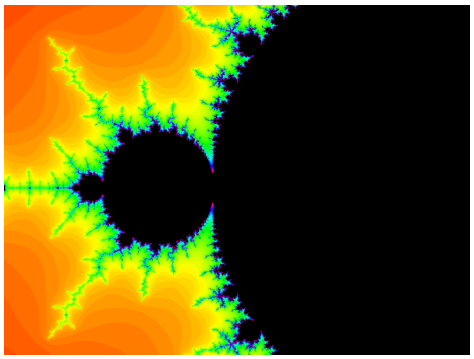

Figure 5.5: An image of the Mandelbrot set with the limits *xmin:* -1.4748333, *xmax:* -0.9748333, *ymin:* -0.1791667, *ymax:* 0.1958333.

Matrix multiplication

The Matrix multiplication implementation uses the same features exploited in the experimental study of Sect. 3.4, using different optimized kernels for each device type.

Black-Sholes

The Black-Scholes case study is the one chosen also in Sect. 3.4. We have explored two case studies using this benchmark: A simple execution of the kernel (BlackScholes), and a program that iteratively launches a sequence of 2048 executions of this kernel for the same array (BlackScholes_2028). As in the matrix addition benchmark, the data transfers are not negligible compared with the computation time. In our model, the same generic kernel definition is used for both CPU and GPU.

5.3.2 Development effort

In this section we compare, in terms of development effort, the use of our library with the most common native programming models for NVIDIAS's GPUs and multi-core CPUs, which are CUDA and OpenMP respectively. For this comparison, we introduce a new development effort metric: Halstead development effort [63]. Thus, we present results for COCOMO lines of code, McCabe's cyclomatic complexity [86], and Halstead development effort. The metrics are applied to the parts of code that include: kernel definitions, kernel characterizations, the coordination code in the main host thread with the Multi-controller management, and data-structure management. We ignore code devoted to error or results checking, performance instrumentation, and writing messages to the standard output.

Table 5.1: Development effort measures for the four benchmarks when they are programmed using Cuda, OpenMP, and the proposed Multi-Controller library.

Benchmark	Code	Lines of Code	Cyclomatic Complexity	Halstead Measure
Matrix	Cuda	72	7	202361
Addition	OpenMP	49	11	99783
	MCtrl	61	5	103528
Matrix	Cuda	142	5	409862
Mult.	OpenMP	45	9	81136
	MCtrl	97	6	201242
Black-	Cuda	211	7	742735
Scholes	OpenMP	134	8	389556
	MCtrl	163	6	486956
Mandel-	Cuda	49	3	159481
brot	OpenMP	36	5	66610
	MCtrl	55	3	124212

Table 5.1 shows the different measures for the different codes evaluated. The results show that our library implies less development effort for the programmer than using CUDA for all the study cases. On the other hand, although the OpenMP programming model needs a less volume of lines of code, deriving in lower Halstead development effort measure, the cyclomatic complexity of our proposal is less because our abstraction hides some run-time decisions and checkings.

Transforming a CUDA program into an OpenMP version, or the opposite, is not a trivial task. Remember the Fig. 5.4, where we show two codes, using the Multi-Controller abstraction, that perform a matrix addition using different target device combinations. When we compare both codes, we observe that the effort required by the programmer to change the program in order to exploit 1 GPU + 10 CPU-cores, or two GPU devices, only involves 4 lines of code. We can see these four lines highlighted on both codes in the code. In a multi-device program written with CUDA and OpenMP, the complexities of both tools are added, while the Multi-Controller abstraction makes transparent to the programmer all the differences on data transfers, kernel launches, and data managements for different kind of computational devices such as accelerators or groups of CPU-cores.

5.3.3 Performance results

In this section, we present performance results: (1) comparing our proposal with pure CUDA reference programs, and (2) evaluating the impact of using in our model different compu-

tational units for the five case studies selected. The goal of this study is to determine the potential performance penalty introduced by using our approach, as well as the performance gain obtained when exploiting a combination of heterogeneous devices with different computational capabilities.

The experiments have been executed on a host machine named *Hydra*, with two CPUs Intel Xeon E5-2609 v3 @1.90GHz, 64Gb DDR3 main memory, and two GPUs: an NVIDIA's GeForce Titan Z (named GPU-0) and a Titan Black X (named GPU-1). We exploit the two GPU devices, and multiple CPU-cores organized in a single virtual device in our model. For this test, we have decided to avoid performance effects derived from oversuscription or hyperthreading. As our Multi-Controller library uses one host thread for each device to be controlled, the number of CPU-cores we use to compute and execute kernels is 10.

The programs have been compiled using the CUDA Toolkit 8.0, and GCC 4.8.3. We have used the flags, *-O3*, and *-fopenmp* to exploit parallelism when using a group of CPU-cores as a computational unit. We have executed all the experiments ten times, registering the lowest total execution times. We have also measured separately the times spent in copying data forth to and back from the target devices, and the computation time of the kernels. We include the time spent by our queue system inside the kernel computation cost.

We have tested three kinds of codes:

- A native CUDA implementation of the different benchmarks tested. We present measures obtained in one or both GPUs in our target system depending on the study. See the right-most and/or the left-most columns in the figures discussed bellow. **Cuda_Ref0**: Measures in NVIDIA's GeForce Titan Black Z, and **Cuda_Ref1**: Measures in NVIDIA's GeForce Titan Black X.

- **CPU+GPU**: This code, programmed using the MCtrl library, executes the programmed application on two devices, a group of 10 CPU-cores, and an NVIDIA's GeForce Titan Black Z. Different mappings have been tested, determined by the percentage of data and computation assigned to each device.

- **GPU+GPU**: This code, programmed using the MCtrl library, executes the application on the two GPUs available in the target system. Again, different mappings have been tested, determined by the percentage of data and computation assigned to each device.

Figure 5.6 shows the performance results obtained by our proposal when the data, and thus the computation, is divided among the group of CPU-cores and the NVIDIA's GeForce Titan Black Z. In the applications where data transfers dominate the total time (Matrix Addition and Black-Scholes benchmarks), we can achieve a better performance by giving part of the computation to the group of CPU-cores. Despite the computational power of the GPU accelerator, the computation division improves performance by reducing the time spent in data transfers to/from the GPU.

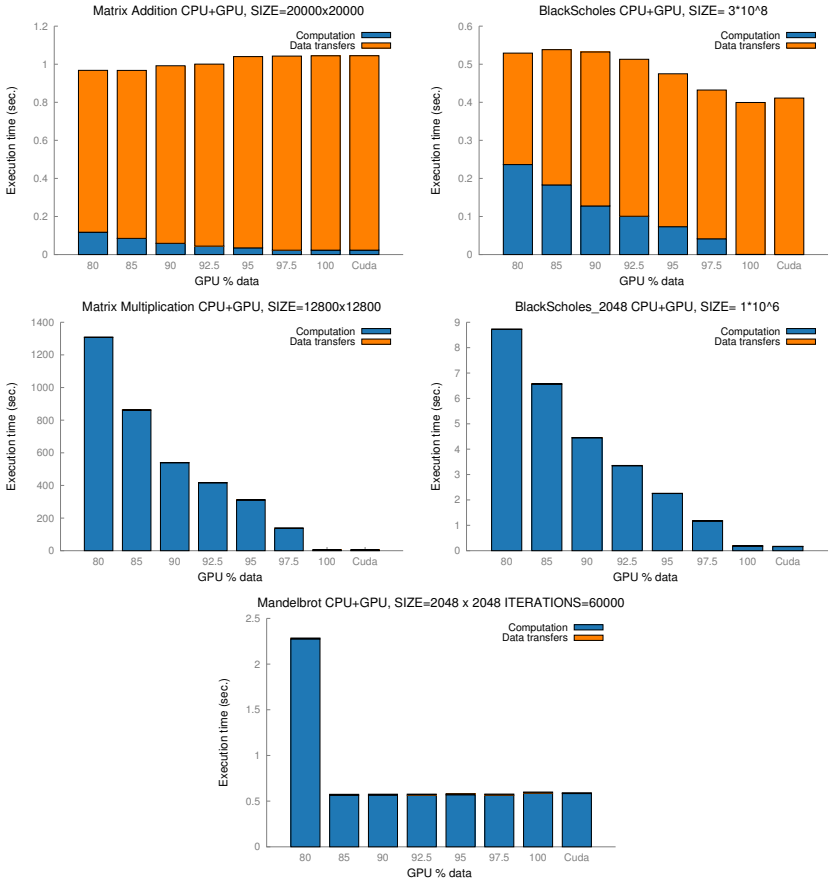

Figure 5.6: Performance results (in seconds) for experiments on Hydra using a group of 10 CPU-cores and a GPU. The right-most columns show the result of the reference CUDA programs run on the same GPU.

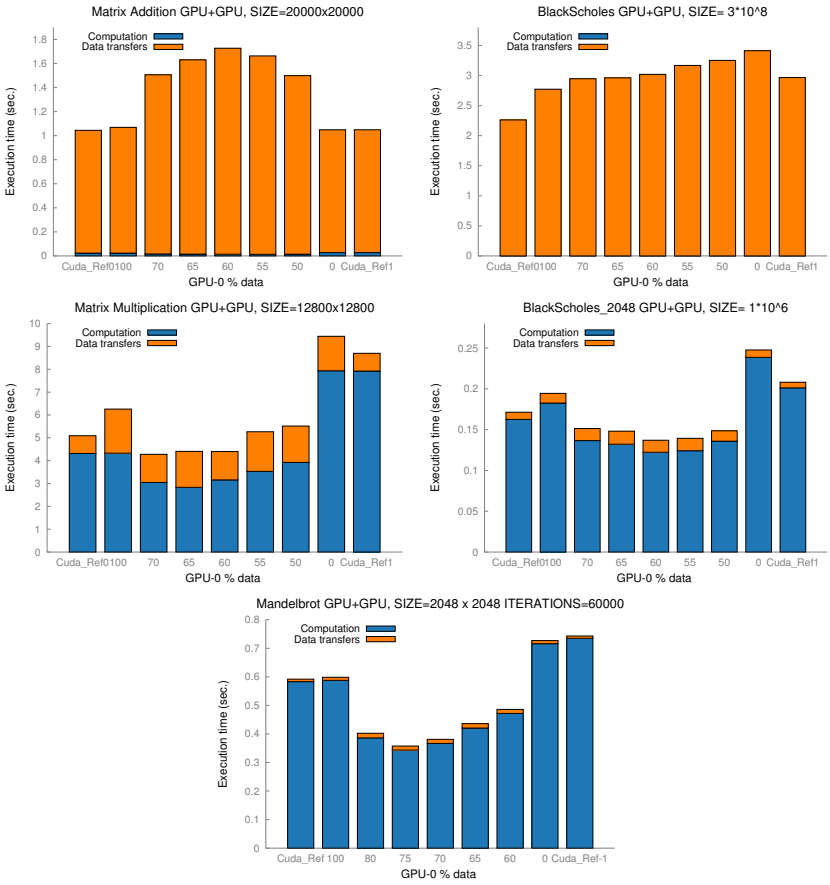

Figure 5.7: Performance results (in seconds) for experiments on Hydra using two different GPUs. The left- and right-most columns show the results of the reference CUDA programs run on each of the two GPUs considered in the study.

When the computational load is high, such as in the matrix multiplication, and Black-Scholes_2048, the best performance is obtained by doing the whole computation in the GPU. This is typical for this kind of programs that really suit the GPU computational model. For the Mandelbrot benchmark, we observe that the partition of the computation obtains marginally better performance results than the CUDA reference codes. This particular behaviour is because of the irregular workload of this benchmark. When the CPU workload increases (case of 80%), some pixels with a high computational cost rely on CPU threads (remember the irregular workload of this benchmark), delaying the execution time and reducing the performance of the application.

Figure 5.7 shows the performance results obtained when we divide the computation between the two GPUs. When the computation load is really low and there are multiple kernel launchings, the time spent in the queue managements (remind that these times are taken into account in the computation time) can be noticeable, such in the BlackScholes_2048 case (where it takes approximately 0.04 seconds).

In our first prototype of the library, data transfers for several GPUs are still sequentialized and not fully optimized, deriving in higher transfer times than the reference implementations. Thus, in those applications where data transfers lead the execution time, the best performance is obtained when the application is executed only in the most powerful GPU. However, when the computation time is much higher than the communication times, a division of the computation among the GPUs, proportional to their relative computation power for this problem, improves the performance. For example, in our study, without taking into account the data transfers, the kernel execution times are reduced the 34%, 25%, and 41% compared to the best CUDA reference codes which use a single device, for the matrix multiplication, Black-Scholes_2048 and Mandelbrot benchmarks respectively. This behaviour is shown in the experimentation results.

5.4 Summary

In this chapter, we presented the Multi-Controller (MCtrl), an abstract entity implemented in a library, that coordinates the management of heterogeneous devices, including accelerators with different capabilities and sets of CPU-cores. This entity offers a global view of the computation, transparently managing the coordination, data partition, mapping, and execution of whole computations on their associated devices. Our solution allows the use of simple generic kernels (portable across different device types), or specialized implementations defined and optimized using specific native or vendor programming models (such as CUDA for NVIDIA's GPUs, or OpenMP for CPU-cores). The run-time system automatically selects and deploys the most appropriate implementation of each kernel for each device, managing the data movements and hiding the launching details. Results of an experimental study with five study cases indicate that our abstraction allows the development of flexible

and highly efficient programs, that adapt to the heterogeneous environment. On-going and future work include studying the support for other kinds of accelerators, and the effect of more sophisticated techniques for data movement that eliminate unnecessary overheads.

Part II

Automatizing the data management for distributed-memory spaces in heterogeneous systems

We presented in the first part of the thesis, a programming model that simplifies the parallel programming on heterogeneous systems, defining a heterogeneous system as a machine with a host and several different accelerators. The separated memory space of the different devices that can compound a heterogeneous system makes necessary the execution of data communications in applications with data dependences. Thus, the Multi-controller approach presented in this first part is valid only for applications where there is no data dependences.

In the next part of the thesis, we present the design, implementation, and evaluation of new techniques in order to calculate automatically the data communications needed for the execution of applications with data dependences in any kind of heterogeneous systems. The extreme cases with separated memory space are the distributed-memory systems, where the communication across devices needs to cross internal node buses, and potentially network infrastructure across nodes. In this second part we present a set of generic and automatic techniques that automatize several parallel programming issues related to distributed-memory

scenarios, making transparent to the programmer issues related to the data transfers, data partition, and data allocation.

Chapter 6

State of the art on automatic management of distributed-memory spaces

In this chapter we introduce the reader to the state-of-the art approaches proposed to tackle the automatic management of data movements and synchronizations for processes executed in environments with disjoint memory spaces, and when data-dependences are an issues. This includes techniques that are used in both, compilers and runtime systems, for code generation, automatic parallelization techniques, and for transparent task communication and synchronization. We analyze the state-of-the art techniques and their implementations, in order to identify their limitations. These limitations define the goals of the part II of this Thesis.

6.1 Motivation and related Work

There are many proposals of abstractions and tools to simplify the management of distributed data and computation for parallel programming. Taking into account the high latencies of accessing remote data and minimizing the number and volume of communications is key. For example, the PGAS (Partitioned Global Address Space) models [126] present an abstraction to work with mixed distributed- and shared-memory environments. The programmer uses the same interface to access local and remote data transparently, but they are aware of the locality of data and can take it into account. One of the most advanced and known PGAS programming models is Chapel [31]. It proposes a separation of domain and mapping modules to generate distributed arrays. However, the best communication aggregation methods presented so far for Chapel abstractions are restricted to specific operations, or domain mapping properties. For example, the work in [118] is restricted to global array assignments with block or cyclic distributions. The work in [119] presents a symbolic substitution of mapping attributes in affine access expressions. It only works for cyclic or block-cyclic distributions. Moreover, the Chapel runtime cannot automatically aggregate several expressions across different loops to generate the full task footprint. It needs to rely on non-aggregate communications when the whole set of data accessed by an expression is not fully allocated in the same remote processor, generating many unnecessary communication overheads in the general case. The programmer can explicitly introduce aggregated communications with an extra development effort. For example, the work [68] presents an approach to manually build aggregated communications, with a low level of abstraction, similar to the one used for UPC [53] or Hitmap [55].

Many task-oriented programming approaches (like StarSs, OmpSs [28, 108], or the works presented in [14, 34, 35, 82]), that also simplify the parallel programming, are based on iterators to generate pools of tasks with explicit input and output working sets. The working sets analysis allows dependence graphs to be built. However, due to the task execution model based on dynamic scheduling with task-queues, synchronizations, and dynamic mapping information, these models cannot derive aggregated communication calculations for groups of tasks statically. Moreover, the use of these task-oriented approaches in distributed memory leads to performance penalties, when comparing to similar static pre-generated and tuned codes, in the general case, due to the task creation and destruction, the management of distributed queues, the synchronization and load balancing mechanisms, and the data communications due to dynamic task scheduling and/or migration. In data-flow approaches, such as the distributed-memory extension for FastFlow [2], the task construction implies a data partition and a dynamic control of task that leads to balanced load. However, this dynamic scheduling prevents the exploitation of affinities and data locality across tasks. Another example is STAPL, STAPL [5] provides, through the usage of the STL library [122], a model of parallelism that supports recursive parallelism and recursive data decomposition, generating a data dependence graph to distribute tasks among the processes ensuring the right execution order.

Many automatic techniques for code generation have the goal of relieving to the programmer to deal with many implementation issues. Classical examples include the works developed in the context of HPF (High Performance Fortran). An example is the dHPF compiler [88]. It applies techniques such as the one presented in [36], that can reduce both communication frequency and redundant data transfer in compiler-generated code for regular, data parallel applications. However, this kind of techniques are applied at compile time by the dHPF compiler in order to generate code. Thus, these solutions are not flexible to use any kind of data distribution policy, specifically those that use non-parametrizable runtime or data-dependent information.

The polyhedral model has been proved to be a useful tool to transform and generate parallel programs from sequential codes with affine nested loops [16] (having a lack of support of other kinds of non-affine applications). It can also be used to automatically generate code for distributed-memory platforms [4]. The dependence analysis supported by the model allows the generation of the code that: (1) identifies which values should be communicated across processes, (2) performs the packing/unpacking of the data, and (3) executes the proper communication operations. All the polyhedral techniques presented so far for distributed memory need to parametrize the iteration space polyhedra and analyze dependences at compile time. Griebl [38] presents a model to use the polyhedral model on distributed-memory systems. However, his technique produces many redundant communication. For distributed memory, the best communication calculation methods so far (see e.g. [18, 21, 42, 115, 143]) compute communications for sequences of arbitrary nested loops with regular (affine) accesses, also known as affine loop nests. The loops are transformed, tiled and finally parallelized. Communications cannot be calculated across different sections of affine loop nests unless loop fusion can be done. These techniques analyze at compile time the footprint of tile data used by other tiles. This implies that the tile size must be fixed at compile time and must be the same for all the machines involved in the computation. Moreover, using these methods, there are still cases for duplicated or unnecessary data communications [21].

There are tools which bring together the advantages of the polyhedral model and the task-oriented programming proposals [80, 81]. These approaches reduce global barriers in shared- and distributed-memory parallel codes by generating tasks with calculated dependences and footprints. However, this approach also performs a data partition after the application of tiling techniques with the tile sizes predefined at compile time [25]. The best tile size depends, among other things, on the architecture details of the target machine where the program will be executed [87]. Choosing tile sizes at compile time prevents automatic tuning for different devices in heterogeneous environments. There are some proposals such as [66] that generate loops that iterate over full rectangular tiles, with unknown parametric tile size. However, it has not been proof that these techniques can be applied with the current communication code generators for distributed-memory systems.

Other similar approaches based on compile-time intersections of parametric polyhedra have been proposed to reduce data transfers in accelerators, such as FPGAS [109], where communications are calculated and optimized only between the host and the accelerator.

Distributed-memory programs introduce the complexity of dealing with data partition policies, and different communication patterns across a number of processes only known at runtime.

The work in [83] presents a hybrid compiler-runtime translator scheme, that calculates the communication pattern needed among SPMD (Single Program Multiple Data) blocks. However, they only support *regular and repetitive* applications where the communication pattern is the same in all the iterations of the outer serial loop that encloses the SPMD blocks. This constraint is also found in other distributed-memory approaches that integrate classical polyhedral techniques for regular codes, with inspector/executor techniques [113] in order to support irregular or indirect data access expressions. This inspector/executor technique exchanges control data before actual communications to avoid traversing the whole iteration space of the parallelized loop on every process.

Fortran-D compiler techniques where presented two decades ago [70] for calculating communication in SPMD programs. They use domain calculations to generate, at compile time, different communication code depending on the data partition selected. This constrain avoids to change data partition features at runtime. However, adapting the data partition based on the details of the target machines is key to achieve a good performance and a balanced workload, especially in distributed-memory systems that include machines with different architectures. Adapting these compiler-based techniques to do part of the calculations at runtime, to take into account details about the platform configuration, partition selection, and input characteristics, introduces a new research perspective on our search for tools and techniques to automatically manage distributed-memory spaces.

6.1.1 Parallel libraries

There are many works that provide external libraries to ease the parallel programming, using distributed data structures with a global view. One of them is DASH [51, 52]. DASH is a realization of the PGAS (partitioned global address space) model in the form of a C++ template library. However, once an array is mapped, it does not provide specific methods for data redistributions.

The STL library has been one of the most parallelized and studied libraries in the literature. For example, the work in [129] presents an implementation for multicore architectures of the STL library using Cilk++. Works like [120, 123] developed parallel versions of this library for shared- and distributed-memory systems using OpenMP and MPI respectively. These approaches only parallelize some selected functions of this library, without providing any programming abstraction out of the STL library scope.

6.2 Summary

In this chapter we analyzed the state-of-the-art approaches proposed to tackle automatic management of distributed-memory spaces.

Some of the limitations that we have found are the following:

- Communication aggregation methods are not advanced enough in the PGAS models. The most sophisticated methods are restricted to specific operations, or domain mapping properties.
- Task-oriented programming models cannot statically derive aggregated communication calculations for groups of tasks. Moreover, in the general case, they have performance penalties due to the task creation and destruction, the management of distributed queues, the synchronization and load balancing mechanisms, and the data communications due to dynamic task scheduling and/or migration.
- In polyhedral model approaches:
  - Communications cannot be calculated across different sections of affine loop nests unless loop fusion can be done.
  - Communication codes are generated based on tiles. Thus, the tile size must be fixed at compile time and must be the same for all the machines involved in the computation.
  - There are still cases for duplicated or unnecessary data communications.
  - Runtime cost in the communication management proportional to the problem sizes.

In the following chapters of the Thesis, we present several techniques that solve some of these limitations.

CHAPTER 7

Analyzing the current limitations of communication code generators

In order to better understand the reach and limitations of existing modern techniques to automatically generate communication codes at compile time, we have carried out a deep study of one of the most advanced and sophisticated method presented so far, in the context of polyhedral model transformations.

In this chapter we study the codes generated by the FOP scheme [42]. Comparing with others, this is the one that reduces more the communicated volume of data, being the generated code also parametric in the number of processes and problem sizes. We present a study of the extra cost introduced at run-time by the generated codes to manage the communications. We do an asymptotic complexity analysis in terms of two main run-time parameters: The problem size N, measured as the number of data elements to be processed, and the number of processors P. Our complexity model highlights some potential scalability limitations in terms of these two parameters (N and P) , in the current implementations of these techniques (Pluto compiler framework [22]). In addition, in this chapter, we identify and isolate one of this limitations, related to the application of the distribution policy used to schedule the iterations of a parallelized loop. We present for deterministic distribution policies, a simple implementation alternative, previously exploited in Hitmap [55], that eliminates this specific scalability problem.

7.1 The FOP communication scheme

The FOP communication scheme [42] is a model for the automatic generation of communication code for distributed-memory parallel programs, in the context of polyhedral code transformations. It is based in data-dependence analysis across tiles, for parallel programs that distribute the iterations of tiled loops. It has been designed to reduce the volume of data communicated across processors, compared with other state-of-the-art systems that also generate automatically communication code for affine-loop nests. It has been also proved that it provides good performance for small and medium sized data sets, and small number of distributed-memory processes [42]. In that work, the authors describe the conceptual approach and the solution design in detail. An implementation of this scheme is included in the current version of the Pluto compiler [22]. In this section, we summarize the main features of the FOP scheme, and some design and implementation decisions to generate the code.

The code dedicated to calculate and execute communications is inserted at the end of distributed loops. The analysis of RAW dependences is done separately for each different data structure (array variable) involved in a distributed loop nest. For a given iteration of a distributed loop (a given combination of loop index values), the FO (flow-out) set is defined as the set of data generated/written during the iteration, that is required/read during the execution of other iterations. At compile time, the FOP scheme determines a *dependences-partition*. That means, a partition of the flow-out set in subsets of target iterations that are located in different tiles. Each subset is treated independently, leading to a different piece of communication code. This partition is application dependent. The generated code for a given partition can have two different flavors. *Multicast* operations that send all the data in a partition to every processor that requires data from it. And *Unicast* operations that issue a different communication for each iteration that requires data from this partition. To avoid sending more than one time the same data to the same processor, unicast operations are only chosen when it is possible to determine, at compile or run-time, that the receiving iterations are all scheduled at different processors. The authors of FOP propose some rules to determine when it is safe to introduce *unicast* operations, being *multicast* the default choice. In this work we will focus on the default and less complex multicast operations.

For multicast operations, FOP introduces one piece of code for each distributed loop, part of the dependences-partition, and array variable. The code uses several auxiliary data structures: (1) One data buffer per processor involved in the computation, to store the data to be sent; (2) One single receive buffer to store all the data received from other processors; and (3) two counters per processor, to store the amount of data to be sent or received to/from each remote processor. Each piece of code contain three stages:

1. Pack: Pack data while identifying target processors. The iterations space of the distributed loop assigned to the local processor is traversed. For each iteration a function, generated with application specific information (σ), is used to identify which other

iterations (and thus, processors) require data from this partition and iteration. The data are packed (copied) into the corresponding output buffers, and the counters that measure the data to be sent to each other processor are updated.

2. Coordination and communication: The interchange of communication sizes across processors, and the issue of the required point-to-point communications. The coordination step is done with the standard all-to-all MPI collective operation. Each processor sends the value of each output-buffer counter to the corresponding receiving process. This interchange avoids the need for traversing the iteration space scheduled on any other processor (doing the same analysis as for packing), in order to obtain the sizes of data that local process expects to receive from each other remote processor. After the coordination step, asynchronous send and receive operations, with a non-zero value in the corresponding counter, are issued for each processor. With all the receive counters available, it is possible to compute displacements to use one single buffer for all the receive operations.

3. Unpack: Unpack received data. The whole iteration space of the distributed loops is traversed identifying which iterations are scheduled on the remote processors for which local process has received data. For each one of these iterations, it is tested if the local processor is one of the receivers of the data for this partition and iteration (again with a function specifically generated for this application). In that case, the data is unpacked from the buffer to the actual array variable.

7.2 Cost model

In this section we present a cost model for the run-time computational effort of the communication management code introduced by the FOP scheme [42]. We use as reference for specific design decisions the codes introduced by the current implementation of Pluto. Our model measures the asymptotic cost of the calculations needed to issue the communications in terms of two run-time parameters: *Number of processors* (P), and *Problem size* (N), measured as the number of data elements to be processed. The model does not take into account the actual cost of the communications, which is dependent on external factors related to the platform and the communication topology. We model only the extra costs introduced by the automatically generated code to prepare and launch the communication activities (calculations to pack/unpack, and other local coordination activities). The objective is to find scalability limitations introduced when applying the scheme, that could be eliminated by new designs or implementation changes.

As commented in the previous section, in this work we will focus on the cost of the lower complexity multicast operations.

```
1 for (t = 0; t < T; t++) {
2   for (i = 2; i < N - 1; i++) {
3     b[i] = 0.33333 * (a[i-1] + a[i] + a[i + 1]);
4   }
5   for (j = 2; j < N - 1; j++) {
6     a[j] = b[j];
7   }
8 }
```

Figure 7.1: Sequential code of the Jacobi-1D benchmark.

We use as example in the discussion of the cost model the Jacobi-1D benchmark from the Polybench [110] (see sequential algorithm in Fig. 7.1). It is one of the simplest example in the Polybench benchmark that needs communication in a distributed environment. An excerpt of the distributed-memory code generated by Pluto is shown in Fig. 7.2.

7.2.1 General cost for a distributed loop

For simplicity of the discussion, let us consider a single distributed loop L, with an iteration index t. Let $D(t) \subset \mathbb{Z}^+$ be the *Domain* of t, that is the subset of iterations that are traversed by the loop index. Let L° be the run-time cost of calculating the communications needed for all the iterations of $D(t)$ scheduled to a given processor. $O(L^\circ)$ is the upper-bound of L°. Constant factors that are application dependent, and are not affected by run-time parameters will be denoted with c_{name}, where *name* will be a single lower-case letter.

Each combination of distributed loop, variable, and part of the dependences partition, leads to one *Instance* of communication code. Figure. 7.2, lines 6 to 39 show the code generated for one communication instance associated to the array variable b in the Jacobi example.

Let c_z be the number of combinations of different array variables, and parts in the dependences partition introduced by the FOP scheme for the loop L, and $O(y^\circ)$ the upper-bound of the cost of one communication instance. Thus, the estimation of the total cost is:

$$O(L^\circ) = c_z \times O(y^\circ)$$

The cost of one instance of communication y° is the sum of the costs of its three consecutive stages previously described: packing, coordination, and unpacking:

$$O(y^\circ) = O(pack^\circ + coord^\circ + unpack^\circ)$$

In the following sections, we model the execution of one instance of code for a generic iteration of the outer loops. We will focus on the outer loops iterations that lead to maximum potential parallelism of the distributed loops.

```
if ((N >= 1) && (T >= 1) && (N >= 4)) {
   for (t2 = -1; t2 <= floord (3 * T + N - 4, 32); t2++) {
     /* Sequential Code */
         .....
     /* End sequential code */
     /* Communications calculation of array b */
     _lb_dist = max (ceild (2 * t2, 3), ceild (32 * t2 - T + 1, 32));
     _ub_dist = min (min (floord (2 * T + N - 4, 32),
                          floord (64 * t2 + N + 60, 96)), t2);
     polyrt_loop_dist (_lb_dist, _ub_dist, nprocs, my_rank, &lbp, &ubp);
     for (t4 = lbp; t4 <= ubp; t4++) {
        clear_sender_receiver_lists (nprocs);
        sigma_b_1_0 (t2, t4, T, N, my_rank, nprocs);
        for (__p = 0; __p < nprocs; __p++) {
           if (receiver_list[__p] != 0) {
              send_counts_b[__p] = pack_b_1_0 (t2, t4, send_buf_b[__p],
                                               send_counts_b[__p]);
     }  }  }
     if (t2 <= floord (3 * T + N - 5, 32)) {
        MPI_Alltoall (send_counts_b, ..., recv_counts_b, ...);
        req_count = 0;
        for (__p = 0; __p < nprocs; __p++)
           if (send_counts_b[__p] >= 1)
              MPI_Isend (send_buf_b[__p], send_counts_b[__p],... );
        for (__p = 0; __p < nprocs; __p++)
           if (recv_counts_b[__p] >= 1)
              MPI_Irecv (recv_buf_b + displs_b[__p], ...);
        MPI_Waitall (req_count, reqs, stats);
        for (__p = 0; __p < nprocs; __p++) {
           send_counts_b[__p] = 0;
           curr_displs_b[__p] = displs_b[__p];
     }  }
     for (t4 = _lb_dist; t4 <= _ub_dist; t4++) {
        proc = pi_0 (t2, t4, T, N, nprocs);
        if ((my_rank != proc) && (recv_counts_b[proc] > 0)) {
           if (is_receiver_b_1_0 (t2, t4, T, N, my_rank, nprocs) !=0) {
              curr_displs_b[proc] = unpack_b_1_0 (t2, t4, recv_buf_b,
                                                  curr_displs_b[proc]);
     }  }  }
}  }
```

Figure 7.2: Excerpt of the communication generated code by Pluto compiler for the array *b* for the Jacobi-1D solver using the FOP scheme

7.2.2 Problem size and number of iterations

The loops parallelized by the polyhedral model tools represent a transformed space of the original loops. The cardinality of the iterations set of a distributed loop is a function of the problem size $|D(t)| = f(N)$, that can be determined in terms of the transformations applied. We are mainly interested in the loops where the cardinality grows asymptotically with N, allowing the exploitation of more parallelism on bigger problem sizes. Some constants are introduced by the transformations that reduce the overall cost. For example, when tiling is applied, the tile size c_t appears as a divisor on N, $|D(t)| = f(N/c_t)$, with an asymptotic upper-bound still in the order of N: $|D(t)| \in O(N)$.

7.2.3 Distribution policy

A *Distribution Policy* function $\Pi : D(t), \mathbb{N} \rightarrow \mathcal{P}(D(t))$ is used to determine the subset of a domain $D(t)$ that is scheduled on a processor rank $p \in [0, P-1]$. The *Inverse Distribution Policy* function, $\pi : \mathbb{Z} \rightarrow [0, P-1]$, maps each index of the domain to the corresponding processor rank. In general, distribution policies try to obtain a good load balance. Thus, we assume that the number of iterations scheduled on each processor is similar: $\forall p \in [0, P-1], |\Pi(D(t), p)| \simeq f(N)/P$. The run-time cost of applying these functions is denoted with Π°, and π° respectively.

The function Π is used to compute the iterations of the loop, scheduled to the local process. In the example code of Fig. 7.2, lines 7, 8 and 9 calculate the lower and upper limits of the iteration space to be distributed *_lb_dist* and *_ub_dist*. These are the inputs for the Π function implemented in the *polyrt_loop_dist* function (line 10). The outputs, *lbp* and *ubp*, are the lower and upper limits of the locally scheduled iterations. The cost of this function is associated with the parallelization of the algorithm. Thus, we do not consider it as a specific cost introduced by the communication calculations.

7.2.4 Packing stage

The packing stage traverses the subset of locally scheduled iterations. See the loop in lines 11 to 18 in Fig. 7.2, that traverses iterations from *lbp* to *ubp*. It has two main contributions to the overall cost that are computed for each iteration considered.

First, each iteration applies a function $\sigma(i)$, specifically generated for each application, to obtain the list of receiving processors. The sigma function contains a constant number of conditionals c_c, dependent on the application source code. Each conditional potentially applies π to obtain the rank of the processor that has a target iteration. Thus, obtaining the target processors for all the iterations scheduled to a processor, is done in $f(N)/P \times c_c \times \pi^\circ$.

Second, each iteration traverses the list of processors to detect the ones that should receive data from the local process. This is done in $O(P)$, with a very small constant c_s, as it executes a simple conditional. See line 15 in Fig. 7.2. The actual packing operation is done only for the processors detected as receivers (condition evaluated to true). The code

for packing data in the output buffers is application dependent and only traverses the data that is going to be sent. However, data are packed (copied) into a different buffer for each receiving processor. Thus, there could be multiples copies of the same data. In the worst case, all processors should receive the same data. This is dependent on the communication structure of the application. For example, neighbor synchronization communications have $O(1)$ number of processors involved for each data subset, while some communications in LU reductions result in $O(P)$ processors involved. Let us model the cardinality of the number of communications with an *h-relation* function $h(P)$. Let c_v be the mean volume of data to be sent by one iteration for the array variable considered. This is typically a constant determined by the application and transformations applied. Thus, the cost of this second part of the packing stage can be estimated with: $f(N)/P \times (c_s \times P + c_v \times h(P))$. The overall cost of the whole packing stage is estimated as:

$$pack^{\circ} = f(N)/P \times (c_c \times \pi^{\circ} + c_s \times P + c_v \times h(P))$$

7.2.5 Coordination and communication stage

The coordination stage includes several actions, see lines 19 to 32 in Fig. 7.2. It starts with an MPI all-to-all collective communication operation to interchange counters. In general, this type of all-to-all communications are assumed to be done in $O(P)$. Then, the actual point-to-point communications needed are launched traversing the processor ranks in $O(P)$. The actual cost of the communications is not modelled for this work, only the preparation and launching activities. Finally, a last loop is executed that also traverses the processor ranks in $O(P)$ for simple bookkeeping operations. We model the overall cost of this stage (without actual communication costs) by:

$$coord^{\circ} = P$$

7.2.6 Unpacking stage

The data received from a processor have been packed following the iteration order. Thus, they should be unpacked in the same order. See lines 33 to 39 in Fig. 7.2. This stage traverses the whole iteration space of the distributed loop (from *_lb_dist* to *_ub_dist* in the example code), using the π function to determine which iterations are scheduled in remote processors. The cost of this operation is modelled with $f(N) \times \pi^{\circ}$.

A second part of the cost appears only for iterations on remote processors from which data have been received at the local process during the communication stage. In the worst case, this condition check, for a given iteration, can be satisfied for all the rest of P processors. However, we can model again the number of iterations that are going to be detected as valid across the whole space with the h-relation function $h(P)$ of the application. Each locally scheduled iteration produces a mean of $h(P)$ communications received from other iterations. For these set of valid iterations, a second check is done using a tailored function that contains

one or more pieces of code (a constant number c_d of them, dependent on the source code), internally applying the π function. Finally, the actual unpack operation is done only once for each data element, and the cost directly depends on the volume of data communicated v. The overall cost of the whole unpacking stage is modelled by:

$$unpack^\circ = f(N) \times \pi^\circ + f(N)/P \times h(P) \times (\pi^\circ + c_d + v)$$

7.2.7 Total cost

Our final cost model is dependent on two functions, and some constants, that should be determined for each application: $f(N)$, $h(P)$, v, c_c, c_d. As we are mainly interested in the asymptotic behaviour, it should not be difficult to determine the order of the functions in terms of N and P. The constants only give us a rough idea of the weight of each part of the formula, but they cannot be considered alone for a really precise model, as the amount of arithmetical operations generated by the loop transformations to access the data elements, pack/unpack them, and similar operations, has not been considered.

The overall cost of calculating a generic communication instance y°, can be estimated as the accumulation of the three stages: $y^\circ = pack^\circ + coord^\circ + unpack^\circ$.

$$\begin{aligned} y^\circ = f(N)/P \times (c_c \times \pi^\circ + c_s \times P + c_v \times h(P)) \\ +P \\ +f(N) \times \pi^\circ + f(N)/P \times h(P) \times (\pi^\circ + c_d + v) \end{aligned}$$

After multiplicative constant factors elimination, and some simplification the asymptotic upper-bound can be modelled as:

$$O(y^\circ) = O(f(N) \times \pi^\circ + f(N)/P \times \pi^\circ \times h(P) + P)$$

7.3 Proposal: Implementation alternative

As it can be observed in the cost model formula, a key operation is the identification of the processor that owns an iteration of the distributed loop. This operation is performed by using the inverse distribution policy function π. It appears several times in the cost model, as a multiplier factor.

Given an unknown distribution policy function Π, a simple way to build π is to execute a loop that applies Π to each processor rank, checking if the iteration parameter is in the resulting set. See pseudo-code in Fig. 7.3 (top left). For example, the run-time Polyrt library version included in the current Pluto distribution, contains only one Π function: A classical block distribution policy. See pseudocode in Fig. 7.3 (top right). With the current Pluto's implementation, the cost of the functions is: $O(\Pi^\circ) = O(1)$, and $O(\pi^\circ) = O(P)$.

```
1
2
3  function pi(Dom d,int i,int P)
4    do p = 0, P-1
5      d' = PI(d,p)
6      if i in d' then return p
7    enddo
8
9
```

```
1  function PI(Dom d,int p,int P)
2    if (p < |d|%P)
3      r.lb = d.lb + (|d|/P)*p + p
4      r.ub = r.lb + (|d|/P)
5    else
6      r.lb = d.lb + (|d|/P)*p + |d|%P
7      r.ub = r.lb + (|d|/P) - 1
8    endif
9    return r
```

```
1  function pi_Alt(Dom d,int i,int P)
2    off = i - d.lb;
3    lim = (|d|/P + 1)*(|d|%P)
4    if ( off < lim )
5      return off/(|d|/P + 1)
6    else
7      return (off-lim)/(|d|/P) + |d|%P
8    endif
```

Figure 7.3: Pseudo-codes of the original π (top left) and Π (top right) functions, and our alternative implementation proposed for π (bottom). $Dom < lb, ub >$ represents a tuple with the lower and upper bound of a contiguous 1-dimensional iteration space. $|d| = d.ub - d.lb + 1$ represents the domain cardinality.

For more generic partition policies the cost may increase, because checking if an index is inside a block range can be done in $O(1)$, but for a generic set of n indexes the search cost is at least $O(\log n)$ if it is sorted, or $O(n)$ if it is is not. In this last case the cost of π could go up to $O(\pi^\circ) = O(P \times N)$.

We propose to use a different approach previously used in Hitmap [55]. In Hitmap, the programmer of the distribution policies is forced to develop plug-ins that include both the direct and the inverse distribution-policy functions separately. In Hitmap, the classical partition policies (block, cyclic, etc.) have implementations where the cost of Π° and π° is quite similar, and it is always $O(1)$. This solution can be exploited for any deterministic distribution policy based on an invertible function. For non-invertible functions the programmer may chose to pay the extra run-time cost factor, or an extra cost in memory footprint. It is always possible to store in an array the index of the assigned processor for all the elements in the iteration space, keeping the $O(1)$ run-time cost for the π function.

We have introduced in Polyrt (Pluto's runtime helper functions) a direct implementation of the inverse distribution policy for block partitions, eliminating a multiplier factor of P in several stages of the communication calculation. See pseudocode in Fig. 7.3 (bottom). The asymptotic impact of this change can be seen in the cost model. After substituting the costs of the π function derived from the current Pluto implementation ($O(\pi^\circ) = O(P)$), and

eliminating the constant multiplier factors, the result is:

$$O(y^{\circ}) = (f(N) \times P + f(N) \times h(P) + P)$$

With our alternative implementation, the multiplier P factors, coming from the π function disappear:

$$O(y^{\circ}) = (f(N) + f(N)/P \times h(P) + P)$$

It is specially remarkable that in the original implementation, the size problem is multiplied by the number of processors during the unpacking stage. In the following sections, we present empirical evidence of the impact of creating a specific π function for each distribution policy Π, directly implementing the inverse function with a cost bounded by $O(1)$.

7.4 Case study: 1-D Jacobi

To show how to apply the cost model, we have chosen the simplest case study, the Jacobi-1D program [67]. This application is a good example to study because: (1) the code produced by Pluto includes only one distributed loop with multicast operations; (2) it has a simple neighbor synchronization communication structure; and (3) it is very easy to find proper approximations for the application-dependent functions. Refer generated code on Fig. 7.2.

7.4.1 Cost model parametrization

The code has been generated using the default tile sizes in Pluto ($c_t = 32$ iterations for any tiled loop). The function that computes the number of iterations in the transformed parallel loop, has two input parameters: $f(N, T)$. Where N is the array size, and T is the number of iterations of the original sequential code before transformations. The formula used to compute the limits of the distributed loop index (line 11 and 33) depends on the value of the outer loop index $t2$ (see lines 7 to 9 in Fig. 7.2). These loops create a pipelined execution. During the program progress, the amount of distributed iterations of the internal loops (loops in lines 11 and 33) grows, it keeps stable for a while, and then decreases. The maximum degree of parallelism obtained in the stable phase is related to the problem size parameters. We have experimentally determined that it can be approximated with: if $(3T \geq N)$, then $f(N, T) \simeq 0.01N$; if $(3T < N)$, then $\lim f(N, T)_{T \to \infty} = 3.125T$. Thus, $f(N, T)$ grows linearly with the problem size parameters $O(f(N, T)) = O(\min(N, 3T))$. For simplicity, let us assume that T is always big enough to obtain the maximum degree of parallelism for a given input array size. Thus, $O(f(N)) = O(N)$.

There are two communications instances, one for array a, and one for array b. Thus, $c_z = 2$. The h-relation function $h(P)$ is typically $O(1)$ in neighbor synchronization applications. Indeed, experimental measures with the generated code for the Jacobi-1D program show that the mean values of the h-relation across iterations and processors are: $\bar{h}(P) \simeq 1$ for the

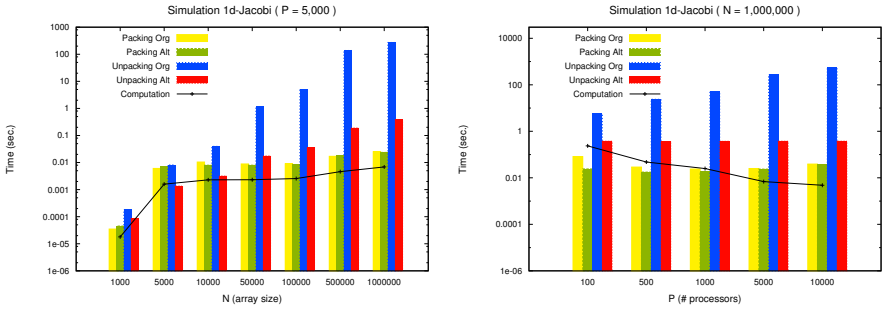

Figure 7.4: Execution times with the original and alternative π function with different problem sizes N and different number of processes P

code instance generated for the array a, and $\bar{h}(P) \simeq 0.25$ for the code instance generated for the array b. The data volume c_v communicated on each distributed iteration has been also measured: $c_v = 188$ data elements for a array, and $c_v = 63$ data elements for b array. Inspecting the generated code, we observe that the other constant values are the following. For the a array $c_c = 7, c_d = 7$, and for the b array $c_c = 1, c_d = 1$.

For an asymptotic behaviour study, we can nevertheless ignore the application constants, and simplify the resulting model for the overall cost of the communications needed for one iteration of the outer loop as:

$$O(L^\circ) = O(N \times \pi^\circ + N/P \times \pi^\circ + P)$$

After substituting the costs of the π function derived from the current Pluto implementation, the result is:

$$O(L^\circ) = O(N \times P + N + P)$$

With our alternative implementation, multiplier P factors coming from the π function disappear:

$$O(L^\circ) = O(N + N/P + P)$$

7.4.2 Simulation study

Doing real experiments for big data sizes, and large number of processors, require a huge amount of computation time in supercomputer infrastructures. Fortunately, we can modify the codes generated to simulate a given amount of the outer loop iterations in a chosen processor, with the desired N and P parameters, using a reduced amount of memory. This allow us to perform an empirical study in order to investigate the effects of scaling the N and P parameters to sizes that resemble high-end supercomputers. Experimental results in a smaller real case and machine are presented in Sect. 7.5.

The modifications needed in the generated code of the Jacobi-1D example include: (1) Add some code to read parameters for the chosen limits for the outer loop ($t2$ index); (2) Change the declarations of the a and b arrays to have a small fixed size (4096 elements); (3) Modify all array accesses to use the resulting index modulo 4096 to stay into the fixed arrays boundaries; (4) Eliminate the MPI calls; (5) At the beginning of each $t2$ iteration, compute locally the send counters for all the remote processors, in order to simulate the all-to-all MPI communication eliminated, that coordinates the communication sizes across processors. This is done out of the code sections that are measured with time counters.

We preserve the same time measuring mechanisms included in the Pluto original code, for the computation section, and for each one of the three communication calculation stages. The data results produced by this simulation are not correct. The communication codes pack and unpack dummy values in the buffers, and in the constricted arrays. However, all the communication preparation calculations, and packing/unpacking operations, are done exactly as in the original code. Thus, the time measures are consistent with the real case, except for the actual communication costs which are intentionally not included or considered in the study.

We discuss results obtained using the simulation program in a PC machine with an Intel-i3 M370 (2.4 GHz) CPU, running a Linux 3.2.29 kernel. The native compiler used is GCC v4.7.1, with the optimization flag $-O3$. We have compiled two versions of the simulation code: One using the original implementation of the π function (Org), and one using the alternative implementation of the inverse function (Alt). The programs are executed with a large range of problem sizes ($N \in [10^3, 10^6], T = N/3$), and number of processors ($P \in [10^2, 10^4]$). The simulation starts at the first iteration of the outer loop $t2$, where the maximum range of the distributed loop (loop with the $t4$ index) is achieved. Then, the code runs 100 consecutive iterations of $t2$. Measures have been replicated with arbitrary processor identifiers $p = 4, 17, 29, ...$, obtaining the same results.

Figure 7.4 shows the measured cost for 100 iterations of the communication code stages of the original (Org) program and the alternative code (Alt), when fixing one of the run-time parameters (N or P). The execution times of the sequential part of the code, that do the actual computation, are also shown with a line. Notice the logarithmic scale on y-axis.

We can observe with the original π function implementation how fast the calculations associated with communication code exceed by orders of magnitude the computation time when the parameters grow. The product of N and P in the unpacking code due to the cost of the π function dominates the cost, growing to more than one minute of clock time for big problem sizes, or a high number of processors.

With our proposed alternative implementation, the P multiplier introduced by the π function disappears. It can be seen in the Fig. 7.4 how the unpacking part of the code is no more affected by it. When the number of processors P grows, the amount of work to be done by each local process is proportionally reduced. Nevertheless, the communication code cost is still dependent on the overall problem size. In our experiments, it exceeds the cost of the computation in one order of magnitude for enough number of processors. We can see

in the Fig. 7.4 how the cost of the unpacking function grows faster than the computation effort for big problem sizes. This is one of the main limitations found in our analysis of the state-of-the-art approaches, for the problem addressed in this part of the Thesis.

7.5 Experimental Study

In this section we discuss a real experimental study performed to verify that the asymptotic behaviour of real codes, when they are executed in real machines, follows the same behaviour than the simulation results, and can be predicted using the proposed cost model.

7.5.1 Experimental environment

We have chosen three study cases included with Pluto compiler as examples, and also included in the Polybench benchmarks. The first one is the already discussed Jacobi-1D program. The second one is a Jacobi-2D program, and the third one is a Floyd-Warshall's algorithm implementation. These programs represent examples of the classes of programs that generate communication code in Polybench. Linear algebra examples in Polybench do not derive in actual communications because Pluto transformations assume that the whole data structures are not distributed, but replicated on each processor, deriving in empty sets of flow-out dependences across processors.

We have compiled two versions of each generated program. One using the original implementation of the π function (Org), and one using the proposed alternative implementation of the inverse π function (Alt). The experiments were executed in a shared-memory machine (Heracles), a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, adding up to 64 cores in total. For this experimentation using a real platform, we have limited the problem size N, and the number of processors P to the maximum supported by the target machine.

7.5.2 Results

Figure 7.5 show the experimental performance measures obtained for the three study cases. We can observe the same predicted results than in the simulation study of Sect. 7.4.2, but in a smaller scale due to the smaller N and P parameter values.

The impact on the performance, achieved by changing the original π function by our proposed alternative, is more noticeable in some problems than in others. It depends on the ratio between sequential computation and communication times. For the three cases of study, the most noticeable effect appears for the Floyd-warshall case, where the packing/unpacking cost is almost 30% of the total execution time, as reported in [21]. This is due to: (1) A higher $h(P)$ factor of this algorithm, comparing with the neighbor synchronization structure of

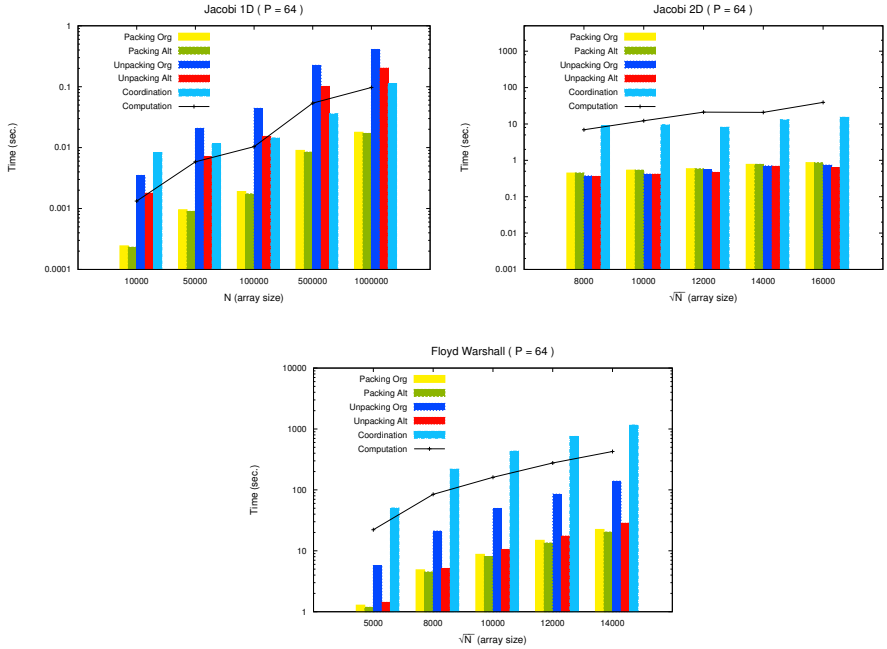

Figure 7.5: Execution times of the codes generated using the FOP scheme, with the original and the alternative π function implementation, for the three study cases: 1D Jacobi, 2D Jacobi, and Floyd-Warshall's algorithm using 64 processors and different problem sizes.

the Jacobi programs; and (2) A higher number of communication instances in the loop. This effect is also predicted by the proposed cost model.

We can also observe that, as predicted by the model, even after applying our proposed alternative implementation of the π function, there is still a proportional increment of the communication calculation run-time cost, with the problem size N. This behaviour is because the FOP scheme relays on checking if communications are needed for the whole space of distributed tiles, on each processor.

Our results show that the cost model is a useful tool to predict the asymptotic behaviour of the code introduced to manage the communications. It can be used to locate scalability limitations when taking design or implementation decisions. We also show how our proposed alternative for the implementation of the π function leads to the elimination of one of these scalability problems. The P and N factors still need a completely different approach to be eliminated.

7.6 Summary

In this chapter we have carried out a deep study of one of the most sophisticated state-of-the-art polyhedral-model mechanism to automatically generate communication codes from sequential codes at compile time. We presented a model for the run-time cost of the FOP scheme. The model allows the study of the asymptotic behaviour of the performance, in terms of the problem size N, and the number of processors P. This cost model highlighted potential limitations of these methods and gives clues to new research directions to solve them, devising new alternatives that do part of the calculations at runtime, when more information about the execution environment and state are available.

Chapter 8

Automatically calculating communications for distributed-memory systems from uniform affine data-access expressions

In this chapter, we present a new communication calculation technique to be applied across different SPMD (Single Program Multiple Data) code blocks, containing several uniform data access expressions. This work tackles the fourth goal:

Design and develop a runtime technique to automatically calculate aggregated communications for applications with uniform affine expressions.

The proposed technique computes at runtime exact coarse-grained communications for distributed message-passing processes. Applying this technique at runtime has the advantage of being independent of compile-time decisions, such as the tile size chosen for each process. We have implemented this technique in Trasgo, a programming model and compilation framework that transforms parallel programs from a high-level parallel specification that deals with parallelism in a unified, abstract, and portable way. We present an experimental study to evaluate the potential overhead introduced at runtime to do the communication calculation, that shows the efficiency of the codes generated with the proposed technique.

8.1 Introduction

As we have seen in the previous chapters, the current state-of-the-art techniques for the automatic generation of communication code for distributed-memory systems present several limitations. In this chapter, we present a new communication calculation technique. Our technique postpones to runtime part of the analysis and decisions needed to transform program abstractions to actual processes. Thus, the programs can adapt their behavior at runtime, dealing with different partitions, granularity, data-distribution, memory hierarchies, tile sizes, or synchronization and communication structures. The technique can be applied across different SPMD (Single Program Multiple Data) blocks of code, that contain several different data accesses expressions to the same data structure, whose indexes are calculate with uniform affine expressions in the indexes selectors. We consider as uniform affine expressions, those expressions that derive in accesses to a multi-dimensional paralelotope of the data structure domain. For two dimensions, this means rectangular shapes. The technique supports codes with several data accesses to the same data structure. Thus, the resulting domain accessed by a code block is a compound of paralelotope shapes, that can be non-convex.

The communications calculated by using our technique outperform those obtained by previous techniques because they are:

- *Coarse-grained* in the sense that communication calculation across two parallel SPMD blocks is done once for the whole index space mapped to a process at runtime, independently of the number or sizes of tiles generated inside the process. This enables different tile sizes to be used in the same computation at the same hierarchical level, an important feature in achieving a good performance on heterogeneous systems that include machines with different architectures [87].

- *Exact* because they are optimal in terms of communication volume. Our runtime calculation skips all the duplicated data elements in the communication. Thus, no data is communicated twice, and no unneeded or control data is communicated across any two distributed processes.

We have implemented this technique in Trasgo [54], a programming model and compilation framework to generate parallel programs from a high-level parallel specification that deals with parallelism in a unified, abstract, and portable way. The proposed technique computes at runtime exact coarse-grained communications between two consecutive parallel blocks for distributed message-passing processes.

We start our discussion with an illustrative example based on a stencil computation in Sect. 2 of this chapter, showing the transformation techniques presented. Section 3 describes the Trasgo model and its tools. Section 4 introduces the new techniques applied in Trasgo. To show the applicability and efficiency of the approach, we include several experimental studies in Sect. 5, comparing performance on distributed- and shared-memory platforms

```
** Illustrative example
Inputs:
   a: 1st stencil parameter
   b: 2nd stencil parameter
   M[n][n]: Matrix with initial values
   limit: Number of iterations
Output:
   M[n][n] : Matrix with result values
Temporal variables:
   M_temp[n][n]: Auxiliar matrix.

** Time loop
Do iter = 1 to limit

   ** First SPMD block: Update M_temp
   Do i = a to n-b
      Do j = a to n-b
         M_temp[i][j] = M[i][j]

   ** Second SPMD block: Compute stencil operation
   Do i = a to n-b
      Do j = a to n-b
         M[i][j] = (M_temp[i-a][j] + M_temp[i+b][j] +
                    M_temp[i][j-a] + M_temp[i][j+b])/4;
```

Figure 8.1: Sequential algorithm for the illustrative example.

with MPI reference codes, and codes generated with auto-parallelizing compilers. The results show that our approach can automatically produce efficient programs despite the overhead of the calculation performed at runtime.

8.2 Illustrative example and Overview

This section presents an illustrative example that serves as a quick overview of the techniques presented in this chapter. The example is a modification of a Jacobi PDE solver for Poisson's equation to compute heat transfer in a discretized two-dimensional surface. This is a simple data-parallel example, that aims to introduce the readers to the basis of the approach. This example can be used to show basic concepts of MPI synchronization (see e.g. [60]). It contains clear computation and communication stages with uniform data access expressions. The sequential algorithm is shown in Fig. 8.1. In our example we also introduce two integer parameters a, b that are used in the access expressions to select at runtime the distance to the positions considered *neighbors* in terms of the stencil operation carried out for that particular invocation of the computation.

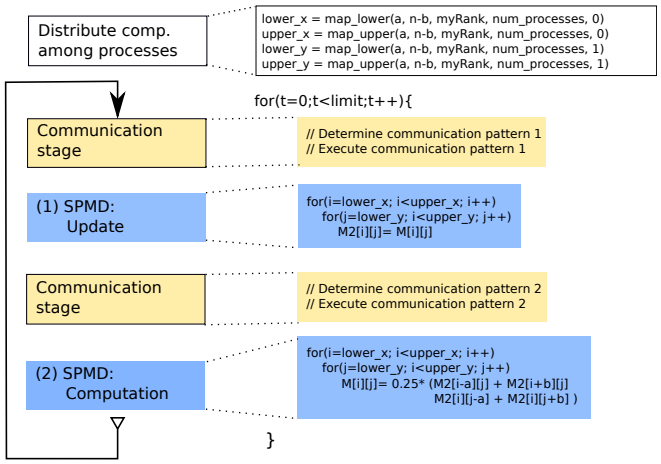

Figure 8.2: Block diagram of the parallel algorithm following an SPMD model for the illustrative example (left), and code excerpts for the main blocks (right). The parameter a determines the halo sizes on the top and left sides, and the b parameter determines the same on the bottom and right sides.

8.2.1 Programming with an SPMD model

We first present an overview of the typical approach used to program the illustrative example following an SPMD model. In the sequential algorithm (recall Fig. 8.1), we can distinguish two different blocks of code inside the time loop that can be parallelized independently without violating any data dependence (transforming them into SPMD blocks). Figure 8.2 (left) shows a distributed parallel programming block diagram of the illustrative example following an SPMD model. To program this algorithm in parallel, the only data dependences that must be taken into account are those produced between the SPMD blocks. For this reason, a communication/synchronization stage is inserted between them.

When we execute this parallel example algorithm in shared-memory systems, a synchronization stage is enough to avoid data dependences between the two parallel structures. However, in distributed memory systems, the data written by a process in the first SPMD block should be sent to other processes that need these data to execute the second SPMD block. Notice that a synchronization between neighbor processes is implicit in this communication.

In our example (see Fig. 8.2, right), we have distributed the computation using the mapping functions *map_lower()* and *map_upper()*. These mapping functions receive the first and last iteration of the loop to be distributed, the identifier of the current process, the total number of processes, and the number of the dimension for which it should calculate the partition

limits. These functions return the loop limits corresponding to the chunk of iterations that should be executed by the current process, avoiding overlappings with other processes.

Each process is ready to perform the computation as soon as it has the corresponding data in its local memory. These data include not only the data in the positions that match the loop iterations, but also the data that correspond to their *halo*, which may be owned by other processes. A communication stage before each SPMD block should retrieve the corresponding halos, thus ensuring that each process has a local copy of all the needed data. In our particular example, before the execution of the first SPMD block, a communication stage is not necessary as each process already has all the data it needs. However, in the second SPMD block, each process needs data that have been updated by other processes (data in the halo). The communication phase in this case depends on execution parameters, such as the matrix size, the tile size, the number of processes, the partition policy (the way in which the data were partitioned among the processes), and the values of a and b parameters which define the halo sizes.

The technique presented in this chapter determines automatically at runtime the communication patterns needed between two consecutive parallel structures, taking into account these parameters, and regardless of their availability at compile or execution time, regardless of the application of other sequential or tiling optimization techniques inside each process.

8.2.2 Overview of the communication determination technique

We present here an overview of the proposed technique to determine automatically the communication patterns among different SPMD blocks. As we have seen in the previous description, we usually need a communication stage between two consecutive SPMD blocks to ensure a correct execution. In our example, we need to communicate some data written in the first SPMD block by each process to other processes that read these data in the second SPMD block. Our technique consists of two steps:

1. In order to determine the data read and written in each parallel structure, for each SPMD block in the program, we generate at compile time a parametrized function that, at runtime, returns the set of data being read or written for a given process identifier p. For the illustrative example, we show an example of both sets of indexes returned by these generated functions in Fig. 8.3. We name the sets of indexes of the matrix M read and written by the k-th SPMD block in the code, at a given processor p as follows: *Input Working Set Indexes* $W_I^{M,k,p}$, and *Output Working Set Indexes* $W_O^{M,k,p}$. These functions are generated at compile time using the data-access expressions found in the input code. In Fig. 8.3, we see how the set $W_I^{\text{M\_temp},2,p}$ is calculated by applying the uniform access expressions found in the code to the calculated loop limits (lower_x, lower_y, upper_x, upper_y). The set of indexes is normalized to be represented with a set of non-overlapped rectangular shapes.
2. In the second step, we apply an algorithm at runtime to determine the communication patterns and to store a compact description of them in an object. To calculate the

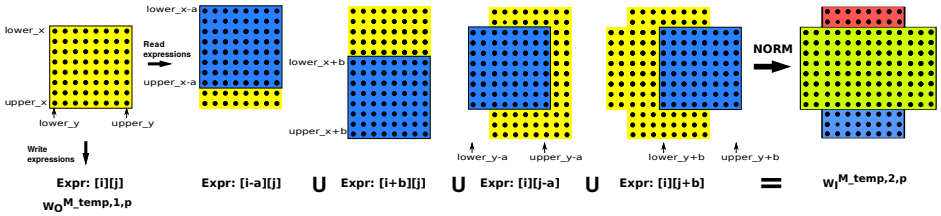

Figure 8.3: Using the read and write data-access expressions inside the parallel structure of the illustrative example to calculate the working input and output index sets (W_I^2,W_O^1) for M_temp at a generic process. This example assumes positive parameters $a, b \geq 0$. The *Norm* operation on the last stage reduces the number of boxes used to represent the union of domains. The drawings consider the particular case of $a = 2, b = 3$, for a domain with 8×8 contiguous indexes.

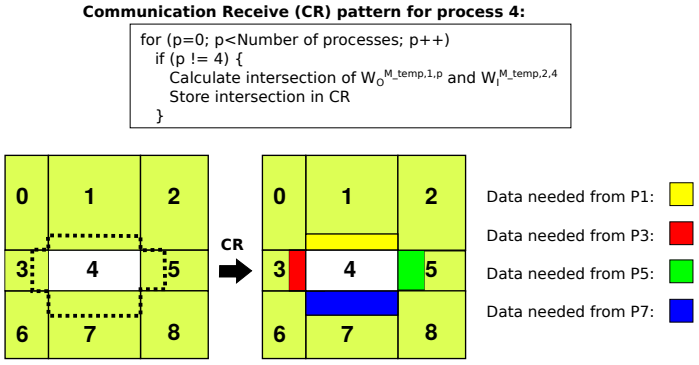

Figure 8.4: Calculation of Communication Receive (C_R) pattern between the two parallel structures of the illustrative example. Example for a number of processes $P = 9$, a process identifier 4, the previously generated functions $W_I^{M\_temp,2,*}$ and $W_O^{M\_temp,1,*}$, a mapping/partition function that returns irregular rectangular blocks, and particular values for symbolic parameters $a = 2, b = 3$.

data that have to be received at a local process from a remote process p, the algorithm intersects the set of indexes read by the local process in the second SPMD block (that is, $W_I^{M,k+1,local}$) with the set of indexes written by the process p in the first SPMD block ($W_O^{M,k,p}$). Figure 8.4 shows a visual representation of our runtime algorithm for the illustrative example. The example shows the calculation of the Communication Receive (CR) pattern for process 4. On the left, we can see the M_temp matrix distributed among 9 processes with an arbitrary irregular partition policy, and the data set (dotted lines) to be read by process 4 in the following SPMD block. On the right, we see the data that should be received by process 4 from different remote processes. The patterns are calculated using the proper intersections of the data space to be read with the data space written or owned by different processes. To calculate the data to be sent by the local process to a process p, the algorithm performs the opposite intersection, between the set of indexes written by the local process in the first SPMD block (that is, $W_O^{M,k,local}$) and the set of indexes read by the p process in the second SPMD block ($W_I^{M,k+1,p}$). An empty intersection indicates that no send (or receive) operation is needed for that particular process p.
After applying this technique for every process and every array we can apply the determined send and receive patterns to perform the actual communications.

Our technique requires that any symbolic parameter must have the same value on every process. Thus, the set of indexes accessed by a remote process p can be calculated by any process, with no inter-process communication. A deeper discussion about the constraints, the features used to reduce the complexity, and a formal definition of our technique is presented in Sect. 8.4.

8.3 The Trasgo Model

The previous section briefly describes our proposed technique. In this section, we review the Trasgo parallel programming and execution model, where our technique has been implemented. The Trasgo model [54] proposes the use of an explicitly parallel, but high-level and structured representation of parallel algorithms. It uses restricted synchronization at the higher level, generating more efficient and less synchronized parallel structures at the low level. The original model is based on the SP (Series-Parallel) process model [134] and data-distribution algebras, providing clear and well-defined semantics [85]. The model is free of race conditions, unexpected dead-locks, or stochastic behaviors. The high-level code uses a global view approach in hierarchical decompositions. The semantics provide clear synchronization points and hierarchical global states that simplify testing and debugging.

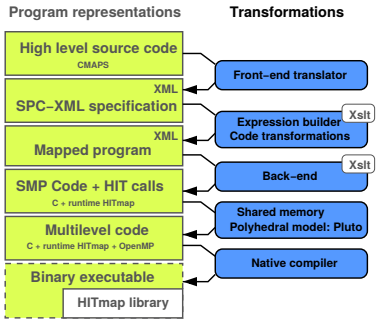

Figure 8.5: Structure of the Trasgo transformation framework.

8.3.1 Overview of the code transformation framework

Figure 8.5 shows the structure of the Trasgo transformation framework. The left column shows the program representations, and the right columns the transformation layers.

A front-end translates the input language (in our case CMAPS [93]) to an XML internal representation. The main part of the transformation layer transforms the global address space into a partitioned address space, building the functions to compute communications across virtual processes. The transformed code is rewritten by a back-end that generates C code with calls to the Hitmap run-time library (remind Sect. 3.3.1). The resulting computation code, generated for the local distributed process, can finally be optimized through polyhedral tools to generate optimized parallel code for the shared memory level using OpenMP. As a proof of concept, we use Pluto [25] on the codes obtained after applying the mapping policy. Thus, we make it optimize the local computation inside each MPI process, in order to efficiently exploit internally the multi-core processor of each node. The methodology used to integrate Pluto with the Hitmap toolchain was described in [95]. The final code is compiled with a native C compiler.

Figure 8.6 shows the illustrative example coded in CMAPS, the current Trasgo input language. First, we define the sequential functions to apply to each element, specifying the output or input role of the parameters (lines 2,8). The parallel function is defined using the *coordination* modifier, and also specifying the role of the parameters (line 13). In its body, the function *ArrayMap()* allocates the distributed arrays M_temp and M in terms of the results of a mapping/partition policy named $partition\_policy$, which is also a parameter (line 18-19). After the distribution, the code updates and computes each element of M_temp in parallel using the sequential functions previously defined (lines 24, 28). The *parallel* structure in CMAPS maps the computation indicated in the *do:* code to each indexes pair in the domain specified in the clause inside the brackets of the *parallel* declaration.

```
/* Second Sequential function */
void updateCell( in double up, in double down, in double left,
                 in double right, out double result) {
    *result = ( up + down + left + right ) / 4 ;
}

/* First Sequential function  */
void updateData( in double Data, out double result) {
    *result = Data ;
}

/* Parallel stencil code with parametric dependences */
coordination
void caseA( inout tile double M[][], in int limit,
            in int a, in int b, in layout partition_policy) {
    double M_temp[][];
    /* Distribute arrays*/
    ArrayMap( M, partition_policy );
    ArrayMap( M_temp, partition_policy );

    /* Time loop */
    loop( t in [1:limit]  ) {
        /* First SPMD block: Update M_temp */
        parallel ( i,j in M) {
            do: updateData( M[i ][ j ], M_temp[ i ][ j ] );
        }
        /* Second SPMD block: Compute stencil operation */
        parallel ( i,j in M ) {
            do: updateCell( M_temp[ i-a ][ j ], M_temp[ i+b ][ j ],
                            M_temp[ i ][ j-a ], M_temp[ i ][ j+b ],
                            M[ i ][ j ] );
        }
    }
}
```

Figure 8.6: CMAPS code for the illustrative example.

8.3.2 Notations and definitions

In this section, we present definitions used in the rest of the chapter. In this work, we focus on arrays with regular dense and strided domains.

Signature is a triplet of integer numbers $S\langle b, e, s\rangle$ (meaning *begin*, *end*, and *stride*). The set of indexes expressed by a signature is $S\langle b, e, s\rangle = \{b \leq i \leq e : (i - b) \mod s = 0\}$.

Domain is a subspace of $\mathbb{Z}^n$. Rectangular n-dimensional parallelotope domains, dense or strided, can be represented by a tuple of n *Signatures*. Let us consider an n-dimensional domain $D_n\langle s_0, s_1, ..., s_{n-1}\rangle \in \mathbb{Z}^n$, where $s_0 \in S^*$, .., $s_{n-1} \in S^*$ are the signatures whose Cartesian product defines the domain. This kind of structures only represent rectangular shapes.

Working set is a generic set of indexes. We represent generic sets as unions of signature domains, $W^M = \cup_{i=0}^{q} d_i : q \in \theta(N), d_i \in D_n$.

Tile is an object that associates data elements of a given *type* to index elements of a domain. The domain of a tile is denoted as $D(T), T : D \rightarrow type$.

Logical process is a tuple $P\langle f, W_I^M, W_O^M\rangle$, where f is a function or subprogram, W_I^M is the working set that P receives as input (the data indexes of M that are read), and W_O^M is the set used as output (the data indexes of M that are written). Logical processes may be composed in sequence, or in parallel. A sequential composition $P_1 \triangleright P_2$ indicates that f_1 is executed before f_2, and that data modifications introduced in the output tiles of P_1 are propagated in the corresponding input tiles of P_2. Sequential composition is associative but not commutative. A parallel composition $P_1 \circ P_2$ indicates that f_1 and f_2 can be executed in parallel. Parallel composition is associative and commutative.

Wave-front composition $(P_1 \bullet (W_f) P_2)$, is a parallel composition with explicitly added order dependences for arbitrary data structures, across processes. If there is overlapping of the shapes of W_f with both the input working set of P_2 and the output working set of P_1, the function f_2 cannot be started until f_1 has finished. The data represented by the overlapping shapes should be propagated. This does not allow generic data-flow compositions to be expressed, with transitive cycles that could lead to dead-locks. This composition operation allows parallel structures, such as wave-front computations and macropipelines, to be expressed when used inside a loop.

Virtual Topology $V(N, R)$ is a graph where the vertices N represent virtual processes, associated with computational resources (groups of processors), and the edges R represent neighbor relations.

Layout $L : D \rightarrow V$ is a function that maps domain subspaces (indexes of tiles or logical processes) to the virtual processes in a virtual topology.

8.3.3 Extensions to the Hitmap library

Several new functionalities have been added to Hitmap to support the new techniques presented in Sect. 8.4. To represent generic domains and working-set indexes in Hitmap, we have created a new data structure, called *HitDomain*. We have implemented functions for efficient domain set operations (intersection $\cap$, union $\cup$, and subtraction $\setminus$) on parallelotope shape structures. A *Norm* function is implemented to transform any domain, represented as a set of signature-based parallelotope shapes, in an union of a set of signature-based parallelotope shapes with empty pair-to-pair intersections. The resulting normalized domain is a set of parallelotope areas with optional stride inside, which can be directly translated to MPI derived data-types for efficient marshalling/unmarshalling operations with no unnecessary data replication. The runtime asymptotic complexity of this function is directly related to the amount of shapes that compose the input set. Also, the actual bound of the q value for a given application (remind that q is the number of signature domains used to compound the domain of a workings set) is directly related with the runtime complexity of applying our

technique. See an example of the use of the *Norm* function in the final step of Fig. 8.3. More advanced representations will be studied as future work.

8.4 Implementation of the technique to determine communication patterns

This section describes in depth the new technique we propose in the context of the Trasgo framework. It allows to determine at runtime the exact aggregated communications across distributed processes, for codes with data-access expressions which are affine transformations of the data indexes used in an SPMD block. Let $i_x : x = 0 \ldots n-1$ be the logical thread indexes in an SPMD block (parallel indexes). The parallel indexes in a CMAPS code are those in the clause inside the brackets of a *parallel* structure. An affine access expression $\rho(x)$ is defined as:

$$\rho(x) = \alpha_0 \times i_0 + \alpha_1 \times i_1 + ... + \alpha_{n-1} \times i_{n-1} + \beta$$

where the coefficients α_x, β can be general expressions using constants and parameters whose values can be unknown at compile time, but are invariant in the body of the parallel structure (SPMD block). In the current prototype we only support uniform affine expressions, whose resulting index domain is a multidimensional parallelotope (hyperrectangular shapes). A uniform expression is defined as:

$$\varrho(x) = i_y + \beta$$

The proposed technique also supports the composition of several blocks that come from the application of several uniform expressions. The resulting domain can be a non-convex domain that is represented by a set of non-overlapped hyperrectangular blocks, representing the exact subspace of the domain that is accessed. See an example on Fig. 8.3.

This section describes in detail the proposed technique. As shown in the illustrative example, we divide our technique into two steps.

8.4.1 Functions to calculate working set indexes

As previously discussed in section 8.2.2, we should generate functions $W_I^k(\ldots)$, and $W_O^k(\ldots)$, which calculate at runtime the input/output working set indexes of each data structure, at each k-th parallel structure (SPMD block). These functions are generated from the expressions found in the *do:* clauses of the CMAPS codes. For parallel structures with *wave-front* expressions, we also generate functions to compute the Input-Flow Working Set Indexes $W_F^k(\ldots)$. The *wave-front* expressions are found in CMAPS in a specific clause that determine the *wave-front* dependences. These functions ($W_F^k(\ldots)$) are generated like other working-set index functions, but using the *wave-front* expressions as if they were read accesses to the data structure.

The generated functions have the following parameters: A process index p, a mapping function $L(p)$ to obtain the subdomain of parallel indexes mapped to p, and the symbolic parameters that appear in the data read/write accesses to the chosen data structure. The code of the function applies, to the index subdomain $L(p)$, all the uniform affine transformations found in the read/write accesses for the data structure, inside the k-th parallel structure. Figure 8.7 shows the functions generated for the working sets $W_I^{\text{M\_temp},2,p}$ and $W_O^{\text{M\_temp},1,p}$ in the illustrative example (they are prefixed by `calcWI`, and `calcWO` in the code). The *HitLayout* objects implement the $L(p)$ methods applied at runtime. The resulting shape domain is transformed according to the code expressions found in CMAPS codes, and the union of domains is computed. ---------- The example of Fig. 8.7, uses specialized Hitmap functions for the case of expressions with only one parallel index on each dimension scope. Let $[\alpha_x * i_y + \beta_x]$ be an expression to access the x-th dimension of the data structure. Notice that the parallel index y does not need to be the same as the dimension that is accessed. The *hit_shapeAffine2(...)* function transforms a 2-dimensional shape to another 2-dimensional shape. For each dimension, the parameters are the identifier of the parallel index y, and the subexpressions α_x, β_x, which are literally copied from the data access expression.

In the current implementation of the Trasgo prototype, we have only taken into account uniform access expressions (According to [143], approximately 84% of the benchmarks of the Polybench [110] can be fully uniformized). The implementation of the transformation functions (such as *hit_shapeAffine2()*) is based on signature (domain) algebras. These functions simply apply the access expressions to the index-space limits at runtime to calculate the resulting shapes. The transformations can be applied independently to each shape, and each dimension. Let us consider a shape domain $L(p) = \langle s_0, s_1, \ldots, s_{n-1} \rangle$. Let $s_y\langle b, e, s\rangle$ be its signature in the dimension y. For an access expression $[\alpha_x * i_y + \beta_x]$, the signature representing the transformed working-set index domain in the x-th dimension can be calculated as $T_1(L(p), y, \alpha_x, \beta_x) = \langle b', e', s' \rangle$, with:

$$b' = min(\alpha_x \times s_y.b + \beta_x, \alpha_x \times s_y.e + \beta_x) \tag{8.1}$$

$$e' = max(\alpha_x \times s_y.b + \beta_x, \alpha_x \times s_y.e + \beta_x) \tag{8.2}$$

$$s' = \alpha_x \times s_y.s \tag{8.3}$$

The transformation functions are applied one by one, according with the data accesses expressions. Their resulting domains are compounded using the *hit_shapeUnion* function. This composition can result in a non-convex domain, that is normalized to eliminate the overlapped parts. The result is a set of non-overlapped rectangular shapes (see Fig. 8.3). These functions are used to calculate the domains, W_I^{M} and W_O^{M}, independently on each process, with no interprocess communication.

Future work includes the implementation of functions to support multi-domains, allowing expressions that involve more than one parallel index. For example, a transformation for expressions such as $[i_0 + i_1][i_0 - i_1]$ would lead to a non-rectangular, rhomboidal shape, that

```
/** Calculate W_I for M_temp in SPMD 2 */
HitDomain calcWI_M_temp_2(HitRank p, HitLayout lay, int a, int b){
   HitShape _TT_mapIdx = hit_layOtherShape( lay, p );           // L(p)

   HitDomain _TT_inWS_matrix = hit_shapeUnion(
     // 2D Affine transformations ( domain, index_x, alpha_0, beta_0,
     //                                     index_y, alpha_1, beta_1)
     hit_shapeAffine2( _TT_mapIdx, 0, 1, -a, 1, 1, 0  ),
     hit_shapeAffine2( _TT_mapIdx, 0, 1, +b, 1, 1, 0  ),
     hit_shapeAffine2( _TT_mapIdx, 0, 1, 0, 1, 1, -a  ),
     hit_shapeAffine2( _TT_mapIdx, 0, 1, 0, 1, 1, +b  ) );

   return _TT_inWS_matrix
}

/** Calculate W_O for M_temp in SPMD 1 */
HitDomain calcWO_M_temp_1(HitRank p, HitLayout lay){
   HitShape _TT_mapIdx = hit_layOtherShape( lay, p );          // L(p)
   HitDomain _TT_outWS_m;
   // No transformations
   _TT_outWS_m = hit_shapeTodomain( _TT_mapIdx );
   return _TT_outWS_m;
}

/** Calculate Communication Pattern: Between SPMD 1 and 2 */
HitPattern calcCommunications_M_temp_1_2( HitTile _TT_Tile1,
                                          HitLayout _TT_lay1,
                                          int a, int b){
   // myRank
   HitRank local = hit_laySelfRanks ( _TT_lay1 );
   // CR,CS = Empty
   HitPattern _TT_patternComm=HIT_PATTERN_NULL;
   // W_O(myRank,L)
   HitDomain WO_L = calcWO_M_temp_1( local , _TT_lay1);
   // W_I(myRank,L)
   HitDomain WI_L = calcWI_M_temp_2( local , _TT_lay1);
   // For p = 0..P
   for (_TT_1 = 0; _TT_1 < hit_layNumActives(_TT_lay1) ; _TT_1++ ){
      // If p != myRank
      if( hit_layToActiveRanks(_TT_lay1, _TT_1) != local ){
         // Send tuple
         // W_I(p,L,a,b)
         HitDomain WI_P = calcWI_M_temp_2( _TT_1 , _TT_lay1, a, b);
         // Intersection
         HitDomain _TT_aux = hit_domainIntersect(WO_L, WI_P);
         // Normalize
         _TT_aux = hit_NormalizeDomain (_TT_aux);
         // Add CS tuple
         hit_patternAdd(&_TT_patternComm,
            hit_comSendSelect(_TT_lay1, hit_layToActiveRanks(_TT_1),
                              &_TT_Tile1, _TT_aux,
                              HIT_COM_TILECOORDS, HIT_DOUBLE) );
         // Receive tuple
         // W_O(p,L)
         HitDomain WO_P = calcWO_M_temp_1( _TT_1 , _TT_lay1);
         // Intersection
         HitDomain _TT_aux = hit_domainIntersect(WI_L, WO_P);
         // Normalize
         _TT_aux = hit_NormalizeDomain (_TT_aux);
         // Add CR tuple
         hit_patternAdd(&_TT_patternComm,
            hit_comRecvSelect(_TT_lay1, hit_layToActiveRanks(_TT_1),
                              &_TT_Tile1, _TT_aux,
                              HIT_COM_TILECOORDS, HIT_DOUBLE) );
      }
   }
   return _TT_patternComm;
}
```

Figure 8.7: Generated code for illustrative example: Functions that build the communications for the parallel structure inside the time loop. Hitmap library is used for tiling management and message passing. Communication is encapsulated on *HitPattern* objects.

can be represented as a set of rectangular shape structures. More complex representations based on octagons [132] can be considered for more general shape representations.

8.4.2 Determining communications patterns

As described in the overview on Sect. 8.2.2, communications should be executed between parallel structures, or between a parallel structure and its next execution if it is inside a loop.

Communication patterns are formed by two subsets $\langle C_R, C_S \rangle$ of comm-tuples (communication tuples). C_R tuples indicate receive operations. C_S tuples indicate send operations. A comm-tuple $\langle p, W^M \rangle$ contains the index of the remote process p, and the working-set indexes W^M of the structure whose data values will be communicated.

Our solution determines the communication patterns needed across SPMD blocks combining two algorithms. Algorithms 1 and 2 are generic models that traverse at runtime the remote active $P-1$ processes intersecting output and input working-set indexes for the same data structure in order to determine the communication tuples needed between two SPMD blocks. Comm-tuples with empty sets, resulting from the intersections, are discarded and not internally stored by the Hitmap objects. For clarity, the data-structure name, the mapping functions $L(p)$ of the indexes at each parallel structure, and the specific parameters are omitted. These algorithms should be implemented on tailored functions for each communication calculation stage, adding the extra parameters needed as inputs in the calls to each specific working-set function.

We distinguish four different cases for using the working-set index functions in order to build the communication constructor functions (See Fig. 8.8). The cases for parallel structures without *wave-front* expressions (namely, (a) and (b)), are simpler. A function is built using Alg. 1. It is tailored to use $W_O^k(\ldots)$ for output working-set indexes. For input working-set indexes, we use $W_I^{k'}(\ldots)$ in case (a), or $W_I^k(\ldots)$ in case (b), which only has a single SPMD block inside a loop. An example of a tailored function to calculate the communication of two consecutive parallel structures in the illustrative example, is shown in Fig. 8.7, with the name `calcCommunications_M_temp_1_2()`.

Our communication calculation and its execution are performed just before the computation of the SPMD block. Nevertheless, if the access-expressions is not dependent on an index of an outer loop, the communication calculations can be inserted as soon as the parameters and L functions are known (even in initialization time in many cases), and the communication execution is inserted before the parallel structure. For example, in the generated main code for the illustrative example (see Fig. 8.9), the invocation to calculate the communications is at line 4 on the main program, before the time loop iteration. Communications are calculated only once, because the a and b parameters are not modified during the computation. The execution of the communications is at line 13, before the second parallel block. It is executed on each iteration.

When the parallel structure k has *wave-front* expressions, we should generate two different communication patterns. The first one is to satisfy the dependences indicated by the

Algorithm 1: Model to calculate the communication pattern across parallel structures, for a given data structure, in terms of intersections of input/output working-set indexes.

Input: P : Number of processes,
$myRank$: Local process id,
$W_O^k(p)$: Function to compute output working-set
$W_I^{k+1}(p)$: Function to compute input working-set
Output: $\langle C_S, C_R \rangle$: Sets of communication tuples

$C_S \leftarrow \emptyset$, $C_R \leftarrow \emptyset$
for *p = 1 to P* **do**
  if *p* $\neq$ *myRank* **then**
    $C_S \leftarrow$
    $C_S \cup \langle p, W_O^k(myRank) \cap$
    $W_I^{k+1}(p) \rangle$
    $C_R \leftarrow C_R \cup \langle p, W_O^k(p) \cap$
    $W_I^{k+1}(myRank) \rangle$
  end
end

Algorithm 2: Model to calculate communication pattern from parallel structure k to itself, after satisfying wave-front flow dependences.

Input: P : Number of processes,
$myRank$: Local process id,
$W_I^k(p)$: Function to compute input working-set of k
$W_O^k(p)$: Function to compute output working-set of k
$W_F^k(p)$: Function to compute input-flow working-set of k
Output: $\langle C_S, C_R \rangle$: Sets of communication tuples

$C_S \leftarrow \emptyset$, $C_R \leftarrow \emptyset$
$W_{tmp1} \leftarrow$
$W_I^k(myRank) \setminus W_F^k(myRank)$
for *p = 1 to P* **do**
  if *p* $\neq$ *myRank* **then**
    $W_{tmp2} \leftarrow W_I^k(p) \setminus W_F^k(p)$
    $C_S \leftarrow$
    $C_S \cup \langle p, W_O^k(myRank) \cap$
    $W_{tmp2} \rangle$
    $C_R \leftarrow$
    $C_R \cup \langle p, W_O^k(p) \cap W_{tmp1} \rangle$
  end
end

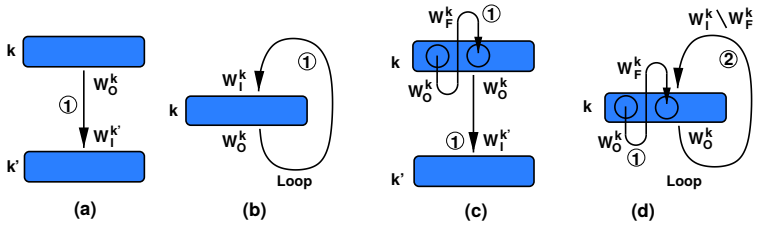

Figure 8.8: Working-set index functions used to tailor the communication constructor algorithms for the four possible situations. Cases (a) and (c) calculate communications across two parallel structures. Cases (b) and (d) calculate communications from one parallel structure to itself, when it is inside a loop. In cases (a) and (b), the parallel structure k does not have *wave-front* expressions. The encircled number represents the algorithm implemented by the generated function.

wave-front expressions across processes during the first SPMD block. In cases (c) and (d), Alg. 1 is tailored to use $W_O^k(\ldots)$ for output working-set indexes, and for input working-set indexes, we use $W_F^k(\ldots)$. The C_R comm-tuples are executed *before* the parallel structure, blocking the process until the dependences from other processes are satisfied. The C_S comm-tuples are executed *after* the parallel structure.

The second communication pattern aims to communicate with the following parallel structure. If it is another different parallel structure, case (c), we use Alg. 1 in the standard way. However, in the case of a parallel with *wave-front* expressions inside a loop (d), we should skip the data already communicated due to the flow dependences. For this particular case, we build a tailored function following Alg.2, using functions $W_O^k(\ldots)$, $W_I^k(\ldots)$, and $W_F^k(\ldots)$.

In the same way than this communication calculation is performed, it would be possible to calculate at runtime the part of the data domain where it is possible to overlap computation with the communication of other parts. The calculation can be performed also at runtime, by operating (intersecting, subtracting, etc.) with the HitShape objects that contain information about the domains of: the data to be sent, the data to be received, and the data to be computed. Future work includes the implementation of this feature in the framework.

8.4.3 Communication patterns for specific applications

In the general case, determining the communication structures involves the comparison of the local working sets with the working sets of the rest of the active distributed processes. Thus, the computation cost of the communication calculation at runtime grows linearly with the number of virtual processes P in the topology. In many applications, the calculation of the communication patterns can be moved out of the loops and computed at initialization time. However, in applications where the communication expressions are parametrized with

```
/* 1. Building comm. pattern A (1 to 2) that is invariant in the
        whole execution */
HitPattern comA;
comA = calculateCommunications_M_temp_1_2(M_temp, M_temp.Layout a, b);

/* 2. Time loop */
for( i=0; i<iterations; i++ ) {

     /* 3. SPMD block: Loops to update M_temp */
     ...

     /* 4. Communication, execute pattern A */
     hit_patternDo( comA );

     /* 5. SPMD block: Loops to traverse the logical processes */
     for ( _TT_i0=0; _TT_i0 < hit_tileDimCard(M,0); _TT_i0++ ) {
        for ( _TT_i1=0; _TT_i1 < hit_tileDimCard(M,1); _TT_i1++ ) {
           /* 5.1. Call functions with selections */
           updateCell(
              hit_tileElemAt(M_temp, 2, _TT_i0-a, _TT_i1),
              hit_tileElemAt(M_temp, 2, _TT_i0+b, _TT_i1),
              hit_tileElemAt(M_temp, 2, _TT_i0, _TT_i1-a),
              hit_tileElemAt(M_temp, 2, _TT_i0, _TT_i1+b),
              hit_tileElemAt(M, 2, _TT_i0, _TT_i1)
           );
        }
     }
  }
```

Figure 8.9: Excerpt of generated code for illustrative example: main program. Hitmap library is used for tiling management, and message passing. Communication is encapsulated on *HitPattern* objects. Reading a, b values from program arguments is skipped.

loop indexes or other parameters, the expressions are not invariant, and the communication patterns should be computed at every loop iteration.

The new Trasgo prototype allows the addition of specialized transformation modules that, by input code inspection, can detect parallelism patterns, and substitute the generic communication calculation by specific optimized functions that do not traverse all the other processes to compute working-set index intersections. The time to compute communications in these cases does not grow with the number of processes. For example, by checking the expressions used in the tile selections, it is possible to detect stencil computations, that derive in neighbor synchronization structures. Similarly, a circular shift pattern is also detectable in Cannon's algorithm for matrix multiplication [30]. Our Trasgo prototype includes modules for some simple stencil and shift patterns, which substitute the generic communication calculation by code that calculates the intersections only with the needed neighbors for both input and output working sets.

These modules to detect specific patterns reduce the calculation communication times. Nevertheless, they cannot be generalized to any dependences pattern or mapping policy chosen. More well-known applications or design patterns can be analyzed and implemented. It is an interesting research question if any well-defined pattern can be detected, and its corresponding communication code can be generated for any mapping policy chosen at runtime.

8.5 Experimental study

We have conducted an experimental study to validate our approach, and to verify the efficiency of the resulting codes. We present four different performance studies:

- One of our main contributions is the ability of our technique to automatically calculate communications on a distributed-memory programming model without a fixed tile size at compile time. For this reason, the experimental study starts with a performance study to show the performance improvement achieved when the tile size is independently tuned at runtime for each machine involved in the computation in a heterogeneous system.

- Second, we evaluate the potential overhead that our runtime technique can introduce when adding more computing elements. The simulation study shows a comparison of the runtime cost of the general communication determination using the described algorithms with respect to using communication patterns for specific applications already included in Trasgo.

- Third, we perform an end-to-end measure, including creation of data structures, data initialization, and the rest of Trasgo potential overheads to compare the programs

generated by our Trasgo prototype with MPI programs manually developed and optimized.

- The last study presents a comparison in terms of computation and communication (determination plus execution) times with a state-of-the-art polyhedral code generator for distributed-memory systems, the Pluto-MPI compiler [21], previously analyzed in Chapter 7, using several benchmarks of the PolyBench [110].

8.5.1 Study cases

The examples and benchmarks discussed, and used in the experimental work are the following. Some of them are Trasgo parallel implementations of programs included in the Polybench [110] benchmarks suite.

Jacobi 2D

This benchmark implements a classical Jacobi PDE solver for Poisson's equation to compute heat transfer in a discretized two dimensional surface. It is implemented as a classical 4-point star stencil code. On each iteration, each matrix position or cell is updated with the previous values of the four neighbors.

We have changed the original 5-point star stencil of the *jacobi-2d* program of Polybench for the 4-point star stencil to compute Poisson's equation. The dependences, and the communication structure, are identical in both cases, but our 4-point stencil presents a slightly lower computational load, making the effect of communications more noticeable, making it more appropriate for our performance comparative studies.

Illustrative example

This benchmark is a modification of the Jacobi-2D program with a fixed number of iterations and two parameters that modify the relative positions of the 4-neighbor points of the stencil. The first parameter a determines the distance to the stencil points on the top and left sides, and the second b parameter determines the same on the bottom and right sides. The data-dependences are dependent on the exact values of the parameters.

Stencil-Opt: An optimized stencil application

This benchmark is another modification of the Jacobi-2D program. It implements a computation window that advances irregularly with the time iterations. The computation window is defined between a *back* column and a *front* column. The back column advances when this column arrives at the stable situation, determined by a convergence condition. The condition is satisfied when the maximum difference of an updated value with respect to the original value in the column cells, at one time step iteration, is lower than a threshold parameter.

Thus, the location of the back column at each iteration is data-dependent. The front column advances one column on each iteration.

Our implementation features a row-based partition. The exact communication needed across iterations depends on the current positions of the back and front columns.

Cannon's algorithm for matrix multiplication

This benchmark is an implementation of Cannon's algorithm for matrix multiplication [30]. It is a task-parallel algorithm for distributed-memory systems. It is designed to reduce the memory footprint at each process. The three matrices are distributed, and no data element is replicated in two processes at any time. Each process holds only a part of each matrix at each iteration. It needs more communication than other approaches that use bigger memory footprints. In order to calculate the communications, we detect the specific shift communication pattern of this application.

Classical matrix multiplication: Matmul

This benchmark is a simple and direct implementation of the classical product of matrices $C = A \times B$, where the three inner loops are parallelized, and each logical process computes and writes the value of one C element. It is an implementation of the *Matmul* Polybench code.

Gauss-Seidel algorithm: Gauss

This benchmark is also a modified version of the Gauss Seidel example in Polybench. We also modified the code to use a 4-point star stencil instead of the original 5-point star stencil. It is a direct modification of our Jacobi-2D example, simply adding *wave-front* clauses to express the dependences on the upper and left elements for each cell (matrix element) across the computation of the parallel structure.

Blur-Roberts

The Blur-Roberts kernel performs edge detection for noisy images. It presents dependences across two different nested-loops which are executed once. We use this example to compare our technique applied on the typical approach used to program this application in distributed-memory systems, with only a communication phase between the SPMD blocks, with a different approach that performs a loop fusion to improve locality, but increases the number of synchronizations.

Table 8.1: Input data sizes ($N \times N$), time loop iterations (T), and threshold parameter, for the different benchmarks in the experimental studies conducted in Heracles and CETA.

Machine	Heracles	CETA
Benchmarks	**Sizes, iterations, threshold**	**Sizes, iterations, threshold**
Illustrative example	N = 7500, T = 200	N = 7500, T = 200
Stencil-Opt	N = 5000, Threshold = 0.001	N = 5000, Threshold = 0.001
Cannon's algorithm	N = 7680	N = 7680
Matmul	N = 4000	N = 4000
Jacobi-2d	N = 7000, T = 1000	N = 5000, T = 800
Gauss-Seidel	N = 7000, T = 1000	N = 5000, T = 800
Blur-Roberts	N = 13000	N = 13000
Gemver	N = 8000	N = 600

Gemver

The Gemver algorithm is a linear algebra kernel that performs a vector multiplication and a matrix addition. It is characterized by a sequence of several parallelizable loop sections with a low computational load.

8.5.2 Experimental platforms and setup

Two clusters have been used in the different experimental studies. The first one is a homogeneous distributed-memory system called *CETA*. It is a hybrid cluster that belongs to CIEMAT and the Spanish government. The cluster nodes are connected by Infiniband technology, and they have two Intel Nehalem-based Xeon 5520 CPUs at 2.27 GHz, with 4 cores each. Using 8 nodes of the cluster, we exploit up to 64 computational units.

The other cluster, *Atlas*, is composed by two multicore machines (*Heracles* and *Zeus*), that acts as a distributed-memory cluster. *Heracles* is a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, and 64 cores in total. *Zeus* is a 6-core Intel E5-2620 v3 at 2.40GHz (up to 12 threads with hyperthreading). Heracles is used in all the experiments to test the scalability of the generated message-passing codes in shared-memory platforms.

In the experimental studies, all the codes, including the MPI reference codes and the programs generated by Trasgo or Pluto, are compiled with GCC v4.8.3 with `-O3` flag. We use *mpich3* v3.1.3 as MPI implementation. For the first experimental study, we use the full Atlas cluster with matrices of $1\,500 \times 1\,500$ data elements. Table 8.1 shows the benchmarks, input sizes of the matrices, threshold, and number of iterations for the rest of experimental studies using Heracles alone and CETA. These problem sizes have been chosen to generate

Table 8.2: Computation times (seconds), of the matrix multiplication benchmark with a size of 1500×1500 on the cluster *Atlas* with different tunning of the tile size. *TS-Heracles*, when applying the best tile size for the Heracles machine in both machines. *TS-Zeus*, when applying the best tile size for the Zeus machine in both machines. *Best TS* represents the program that chooses different tile sizes for each machine, the best for each one.

Processes	TS-Heracles	TS-Zeus	Best TS
1+1	11.32	6.23	6.23
2+2	5.69	3.17	3.17
4+4	2.87	2.24	1.78
6+6	2.00	2.11	1.75
12+12	1.16	1.28	0.99

enough computational load to obtain significant results for our experimental platforms. In some cases, like Gemver in CETA, the size is the maximum supported in the chosen platform by the distributed version of Pluto-MPI, that replicates the memory footprint of the whole global data structures for each process.

As the focus of our work is the efficient automatic calculation and execution of communications among processes, we have launched, in all the experiments, a distributed process for each computational unit, without exploiting the shared memory of the machines. All the results presented are the minimum execution time on ten repetitions of each experiment, to eliminate the outliers produced by stochastic delays in the communication systems.

8.5.3 Improvement achieved by tuning the tile size for each process

We have developed an experimental study to show the positive impact on the performance of tuning the tile size at runtime for each machine involves in the execution, based on their details [87].

We use as benchmark the classical algorithm for matrix multiplication (in CMAPS). We have executed the program with different tile sizes in the two different machines of the Atlas cluster, to empirically determine which is the best tile size for each machine. Then, we execute the program distributing the processes across both machines at the same time, using each process the best tile size for its machine. Table 8.2 shows the runtime execution times when the matrix multiplication is executed (1) using the best tile size found for Heracles in both machines (TS-Heracles), (2) using the best tile size found for Zeus in both machines (TS-Zeus), and (3) using, on each machine, its best tile size (Best TS),

We observe that the best performance is achieved when the tile size is tuned for each machine independently. Unlike previous techniques, our solution does not analyze the

Table 8.3: Execution time for the communication determination for Jacobi-2D solver (seconds).

Processes	General Model	Specific Model	Hierarchical mapping policy
256	2.27×10^{-3}	3.08×10^{-5}	7.60×10^{-4}
1 024	3.92×10^{-3}	3.12×10^{-5}	1.48×10^{-3}
16 384	0.12	2.97×10^{-5}	1.54×10^{-3}
262 144	2.41	3.27×10^{-5}	2.38×10^{-1}
1 048 576	53.22	3.21×10^{-5}	9.53×10^{-1}

index domain nor generates the full communication code at compile time. This allows the adjustment of the tile size at runtime using different approaches. For example, fixing a parametric tile size in an already tiled input code or using works, such as [111], that provide different ways to choose the best tile size at runtime. Our method allows the application of this kind of optimization techniques without changing the communication codes. This feature is not found in other techniques of the related work, where the tiling is also used as a main feature to generate the communication code, and thus the tile size cannot be changed at runtime.

8.5.4 General communications model vs. patterns for specific applications

In this study, we evaluate the potential overhead that our runtime technique can introduce when the number of processes is really high. Each process computes its communication structure independently. Thus, we can isolate and run the code to compute the communication structures on a single process with the proper parameters to simulate the calculations that would be done if the application were launched with any given number of processes. This allows us to obtain accurate measures of the cost of the communication determination alone for a huge number of processes, not actually available in the experimental platforms we have accessed. This study has been carried out with one MPI process in Heracles.

Table 8.3 shows the execution times obtained to compute the communications structure of the Jacobi-2D solver. It compares the actual time to calculate communications using the general model, described in Sect. 8.4.2, with the optimized pattern described in Sect. 8.4.3. Moreover, we also compare the time spent to calculate communications using the general model when a QuadTree hierarchical mapping policy is used to distribute the data. This kind of mapping policies define the distribution of data in several levels (see Fig. 8.10). Our technique is performed at each hierarchical level recursively. The calculation operations (such as intersections or subtractions among domains) are only performed in the next lower/finer level for the parts of the current level whose intersection with the local accessed domain is not

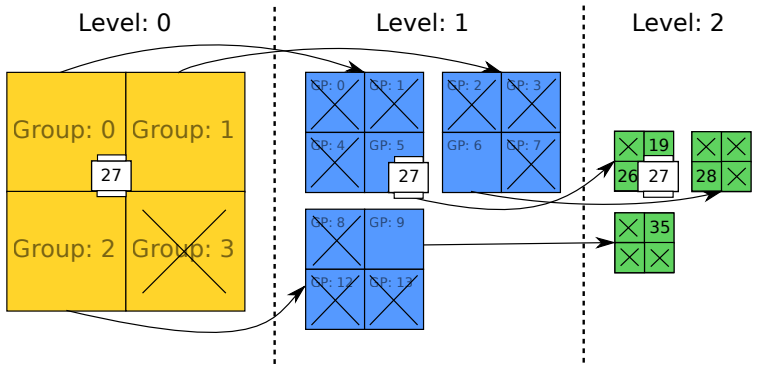

Figure 8.10: Application of the proposed communication calculation technique when using a hierarchical QuadTree mapping policy to distribute a matrix on 64 processes. White rectangular shapes represent the W_I of the 28-th process (id=27) for the Jacobi-2D benchmark. Crosses point out the processes or groups of processes whose mapped data do not intersect with the locally accessed domain at a given level, detecting that no communication is needed. Thus, they are not checked at lower levels.

empty. Thus, in the example shown in the figure, using 64 processes, instead of comparing with all the processes, the comparison is only performed with 4 domains at level 0, 12 at level 1 and 12 at the last level. Using this kind of hierarchical mapping techniques, we obtain a calculation time bounded by $O(log(P))$ for patterns which imply communication with a constant number of processes.

The time of the general model grows linearly with the number of processors as expected (P). For the Jacobi-2D example, we can see on Tab. 8.3 that it is less than two seconds, even for hundreds of thousands of processors. But it may be a hindrance for millions of processors, or if the pattern needs to be recomputed at different iterations of a loop. In these cases, a hierarchical decomposition and nested parallel structures can alleviate the problem, as we can see in the third column of the table. This simulation verifies that the communication calculation time when using the general model on a hierarchical mapping policy, is reduced to be proportional to $log(P)$ instead of P.

On the other hand, the specific pattern for the Jacobi-2D stencil always computes neighbor synchronization pairs, independently of the data sizes or the number of processes. The time in this case is bounded and negligible.

The use of hierarchical mapping policies is an interesting feature to scale the use of the proposed technique to really huge amounts of processes, with the target of exascale computing in mind. However, for the small sizes of the machines used in the rest of our current experimental studies, we use the general model with a simple one level mapping

Table 8.4: Performance (in seconds) obtained for the three benchmarks chosen. For each case we show results for the MPI reference version, and for the Trasgo generated code. Cannon's algorithm by design requires a number of processes with a perfect square root.

	Illust. case		Stencil-Opt		Cannon's	
Machine	**MPI**	**Trasgo**	**MPI**	**Trasgo**	**MPI**	**Trasgo**
Heracles-4	14.48	18.04	163.47	196.86	174.31	186.01
Heracles-8	17.51	20.28	116.76	118.68	-	-
Heracles-16	9.79	11.07	57.34	67.43	56.66	62.94
Heracles-32	4.42	5.43	35.83	43.61	-	-
Heracles-64	4.00	5.08	31.62	35.89	16.10	17.95
CETA-4	25.22	27.69	147.79	166.72	173.46	173.71
CETA-8	22.68	23.20	108.94	114.01	-	-
CETA-16	11.03	12.72	100.97	111.19	48.89	58.46
CETA-32	6.04	6.64	80.11	86.08	-	-
CETA-64	3.37	3.63	57.54	60.20	18.54	21.37

policy, because determining the communication patterns for these cases has an unnoticeable impact on the performance, for any model or application.

8.5.5 Comparison with MPI references

In this study, we compare the programs generated by our Trasgo prototype with MPI programs manually developed and optimized. We perform an end-to-end measure, including creation of data structures, data initialization, and the rest of Trasgo overheads. For this study we use: the Illustrative example; an implementation of Cannon's algorithm for matrix multiplication [30], which is specially devised for distributed-memory systems in order to minimize the memory footprint; and the optimized stencil computation (Stencil-Opt) whose communication pattern is data-dependent and should be recalculated on each time iteration of the stencil program.

Table 8.4 shows the performance obtained by the MPI reference versions, and the Trasgo generated programs for the cases of study. We execute the applications on both, shared-memory and distributed-memory machines. We see that Trasgo programs scale quite well, but losing some performance in comparison with the manually optimized MPI codes (less than 20% in the worst cases). This shows that the implementation of the proposed technique in Trasgo produces efficient parallel programs at the communication level, with a small overhead.

8.5.6 Comparison with a state-of-the-art tool

In this study, we have performed an in-depth comparison of the performance between the codes generated by Trasgo, and those generated for distributed-memory by Pluto-MPI (distmem), the polyhedral model compiler that includes state-of-the-art techniques for generating communication code [42] and that was deeply studied in chapter 7. We choose Pluto-MPI because: (1) According to the authors, it is the first work that reported an end-to-end fully automatic distributed-memory parallelization and code generation for input programs and transformation techniques; (2) It is a free available tool easy to install, which supports all the benchmarks tested; (3) Many research works have appeared that use Pluto as baseline, for both shared- and distributed-memory systems, such as [13, 80, 81]; (4) To the best of our knowledge, the methods of Pluto for code generation in distributed-memory systems, comparing with others, are the ones that reduce more the communicated volume of data, being the generated code parametric in the number of processes and problem sizes.

We have selected five examples from Polybench [110], a collection of examples to be used for testing and developing polyhedral model compilation techniques. The examples are Jacobi-2D, Gauss-Seidel, Blur-Roberts, the classical Matrix Multiplication algorithm and the Gemver algorithm. We have slightly modified the stencil in the Jacobi-2D and the Gauss-Seidel examples. Instead of computing a 5-point star stencil, we compute the 4-point star stencil of Poisson's equation. It reduces the computation load per process, leading to a slightly bigger impact of the communications, which is the focus of this study. Gauss-Seidel has been selected because it presents *wave-front* dependences, deriving in a macropipeline computation. Matrix Multiplication is a good case for simple linear algebra problems with a nice computation/communication balance. On the other hand, Gemver presents a communication/computation balance that is not adequate to distributed-memory systems (communication time typically is higher than computation). The Blur-Roberts filter is a kind of stencil with a single iteration. They were chosen to show the performance of this kind of applications when the techniques discussed in this chapter are applied.

We compile the codes with the *Makefiles* provided by default with Pluto-MPI, which include a *distopt* option enabling the use of the FOP communications model [21], and a set of default flags and tile size values for each example. The *Makefiles* for the stencil examples do not include the flag *-l2tiles* to enable multi-level tiling. The internal tools used in Pluto-MPI do not handle it in an affordable compilation time.

For time measuring, in this study, we have taken into account only the main computation and communication times, named *Seq.* and *Comm.* in the result tables. In Pluto-MPI, the full matrices are allocated and initialized in all processes, although only the parts needed for the local computation are used. At the end of a parallelized affine loop nest, Pluto needs to create again a common global state communicating local results for each process. We have skipped all these times in our Pluto-MPI measures. For fair comparison, we measure as communication times only: The cost of packing and unpacking the data, the cost of communicating control information needed for the communications (only in Pluto-MPI, our proposed technique avoids this information exchange), and network latencies and syn-

Table 8.5: Maximum variation in the execution times for each benchmark in Heracles and CETA, when using Trasgo and Pluto. The variation ratio is defined using the following formula: $((max\ time - min\ time)/min\ time)$.

Machine	Heracles	
Benchmarks	**Trasgo**	**Pluto**
Matmul	0.0231	0.0575
Jacobi-2d	0.0490	0.0707
Gauss-Seidel	0.0079	0.0160
Blur-Roberts	0.2670	0.2591
Gemver	0.1192	0.1965

Machine	CETA	
Benchmarks	**Trasgo**	**Pluto**
Matmul	0.0262	0.0260
Jacobi-2d	0.0196	0.0177
Gauss-Seidel	0.0119	0.0056
Blur-Roberts	0.1695	1.1790
Gemver	0.5304	0.4878

Table 8.6: Main execution times (in seconds) for the five benchmarks chosen from the Polybench.

	Jacobi-2d		Gauss-Seidel		matmul		Gemver		Blur-Roberts	
Machine	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**
Heracles-4	142.65	101.67	428.36	274.04	41.34	28.46	0.65	0.75	0.38	1.36
Heracles-8	83.35	77.74	253.40	196.52	23.23	14.41	1.15	0.76	0.29	1.49
Heracles-16	46.61	58.26	142.41	156.02	13.24	8.26	1.60	0.78	0.24	1.75
Heracles-32	24.18	59.47	113.21	138.58	7.23	4.67	1.85	0.83	0.14	2.05
Heracles-64	18.51	61.32	64.91	128.11	4.05	2.39	2.52	0.86	0.11	2.77
CETA-4	53.44	38.20	122.13	72.08	26.19	30.31	0.0025	0.0049	0.48	1.61
CETA-8	37.03	28.73	71.08	59.87	13.33	14.36	0.0999	0.0046	0.38	5.67
CETA-16	24.28	33.20	62.86	44.93	7.84	12.73	0.1291	0.0787	0.41	20.88
CETA-32	14.62	21.13	40.25	46.65	7.05	6.84	0.4696	0.1673	0.35	47.05
CETA-64	14.56	24.36	35.09	70.30	7.56	4.13	0.4641	0.1310	0.20	59.78

chronization waits. In the computation parts, we have selected the same tile sizes on each example for both, Trasgo and Pluto codes. In addition, in order to provide a measure of the stochastic delays, we also show in Tab. 8.5 the maximum variation of the execution times, for the different benchmarks, for each different execution platform, and for each different tool tested, Trasgo and Pluto. The maximum variation is represented as a ratio of the time difference between the minimum and the maximum execution times. We observe that in the applications Blur-Roberts and Gemver, where the communication time is much higher than the computation time, the variation of the execution times is really high because of the stochastic delays of the network. In order not to take into account this stochastic effects of the network infrastructure, in the rest of tables, we show the minimum execution time achieved in the experiments.

In Table 8.6, we present the total execution times (*Seq+Comm*) considered for each benchmark. In Table 8.7, we present independently the accumulated time expended in the com-

putation stages, and the accumulated time expended in the communication calculation, execution, and synchronization waits. Each result is the measure obtained for the process that expended more time in the corresponding stages. Thus, we can observe in which cases the communication cost is higher or lower, independently of the computation code.

The results for the Jacobi-2d, and the Gauss-Seidel examples indicate that the Pluto code is more efficient for a low number of processes, although it does not scale as well as Trasgo. The transformations performed by Pluto, including skewing the time loop, derive in a lot of re-utilization and exploitation of memory hierarchies inside the processes. Trasgo codes exploit only spatial parallelism at the distributed-level, as in classical manual message-passing approaches. This derives in coarse-grained communications, but fewer opportunities to exploit computation code optimizations inside the processes due to extra synchronizations. It is specially noticeable for the macro-pipeline structure from which Gauss-Seidel derives. However, as the number of processes grows, Pluto reveals its more clumsy communication calculations, while the granularity of the Trasgo communications decreases, and its reduced costs for communications become much more relevant. See the communication cost in Table 8.7 for these examples.

In the case of Pluto codes, the full matrices are allocated and initialized in all the processes. On the other hand, Trasgo use actually distributed arrays, with much lower memory footprint. However, Trasgo needs a communication stage before the execution of each SPMD block. In the case of the matrix multiplication, there is only one execution of an SPMD block. Thus, the communication times in Table 8.7 for Trasgo only include the time to redistribute the data needed for each process to compute its local part. On the other hand, Pluto does not need a communication stage beyond the global state consolidation, thus we do not consider this time spent by Pluto in our study. The communication times for the matrix multiplication in Pluto codes in Table 8.7 are due mainly to the synchronization times to exchange control information in order to determine that no communication is necessary for any process.

As expected, the Gemver example shows poor scalability for both Trasgo and Pluto distributed-memory programs. The computational load is really low, with several high-volume communication stages. The cost of executing the communications is higher than the computation. The sequential algorithm in the Gemver benchmark is not a good candidate for distributed-memory programming in general. The performance in this case can be improved in both approaches using a different mapping policy to distribute the computational load among the processes. However, this study is beyond the scope of this work.

The Blur-Roberts filter is a kind of stencil program with a single iteration invoking two SPMD blocks. The results show that the transformation of SPMD blocks into an affine loop nest performed by Pluto sometimes implies poor performance, specially in distributed-memory systems, due to the need for multiple communication stages in the generated pipeline (although locality is improved). This effect is highly noticeable for CETA, the distributed-memory cluster (see *Comm.* times in Table 8.7). Using solutions such as diamond-tiling [24] can alleviate this problem in Pluto. On the other hand, Trasgo issues a single communication stage for each SPMD block, with the expected scalability.

Table 8.7: Performance (in seconds) of Polybench codes, generated for distributed-memory by Trasgo, and by Pluto-MPI, broken down into computation and communication times (including calculation and execution).

	Jacobi-2d				Gauss-Seidel				Matmul			
	Trasgo		Pluto		Trasgo		Pluto		Trasgo		Pluto	
Machine	Seq.	Comm.	Seq.	Comm.	Seq.	Comm.	Seq.	Comm.	Seq.	Comm.	Seq.	Comm.
Heracles-4	141.94	2.24	85.35	49.43	226.32	283.24	192.41	116.34	41.24	0.09	28.46	2.73
Heracles-8	80.56	16.51	52.52	54.18	129.52	182.18	100.02	123.70	23.05	0.18	14.41	2.69
Heracles-16	45.93	15.24	30.04	48.48	70.61	108.35	53.80	122.33	13.05	0.20	8.26	3.30
Heracles-32	22.46	7.83	22.23	54.78	52.88	95.59	33.25	123.64	7.05	0.18	4.67	3.49
Heracles-64	15.57	7.86	22.96	63.83	27.73	55.60	22.11	123.51	3.79	0.26	2.39	2.39
CETA-4	53.04	8.09	36.88	15.98	62.93	75.75	55.65	29.37	26.12	0.07	30.31	2.62
CETA-8	36.17	13.19	22.15	17.87	32.89	52.95	28.60	36.43	13.22	0.13	14.36	2.66
CETA-16	20.24	13.14	16.49	30.66	22.83	53.43	14.39	35.78	6.97	0.89	12.72	6.16
CETA-32	8.63	9.40	8.54	22.41	19.62	36.24	10.79	43.36	5.32	1.79	6.81	5.85
CETA-64	5.77	12.29	8.55	25.39	8.81	33.09	9.35	70.47	5.60	1.69	4.07	4.13

	Gemver				Blur-Roberts			
	Trasgo		Pluto		Trasgo		Pluto	
Machine	Seq.	Comm.	Seq.	Comm.	Seq.	Comm.	Seq.	Comm.
Heracles-4	0.1816	0.4648	0.7514	0.7538	0.35	0.03	0.70	0.93
Heracles-8	0.1607	0.9864	0.7553	0.7585	0.21	0.06	0.47	1.41
Heracles-16	0.1417	1.4545	0.7790	0.7848	0.19	0.07	0.41	1.78
Heracles-32	0.0504	1.7966	0.8158	0.8286	0.08	0.05	0.39	2.19
Heracles-64	0.0383	2.4794	0.8442	0.8679	0.09	0.05	0.39	3.25
CETA-4	0.0010	0.0019	0.0026	0.0033	0.38	0.10	0.75	1.01
CETA-8	0.0005	0.0989	0.0031	0.0045	0.38	0.15	0.74	4.92
CETA-16	0.0003	0.1203	0.0037	0.0462	0.22	0.33	0.70	19.48
CETA-32	0.0003	0.3767	0.0036	0.1987	0.11	0.30	0.37	45.20
CETA-64	0.0002	0.4633	0.0035	0.1967	0.05	0.16	0.41	53.93

We conclude that Trasgo codes scale very well due to their very efficient communication structures, despite the fact that the computation code can still be optimized further.

8.6 Summary

This chapter presents a technique that, for parallel structures with uniform affine expressions on data accesses, automatically determines at runtime ad-hoc communication patterns for distributed-memory processes across two consecutive SPMD blocks. This new technique uses the results of a partition policy to compute at runtime exact coarse-grained communication patterns for distributed message-passing processes. It is based on intersections of remote and local footprints in terms of the results of the mapping function chosen. Our approach allows the automatic generation of pre-compiled multi-level parallel libraries or programs, that can adapt their communication and synchronization structures to the target system. Experimental results, for several representative cases of study, show that our technique produces efficient codes, despite the overhead of our runtime communication calculation, compared with a compile-time state-of-the-art tool that generates communication codes, and with manually implemented and optimized pure-MPI references codes.

Future work includes the applicability of the transformation model in the context of current polyhedral model frameworks, using more irregular domains, or extending it for non-completely affine expressions.

Chapter 9

Automatically calculating communications for distributed-memory systems from uniform affine expressions on periodic domains

Many real-world applications feature data accesses on periodic domains. Manually implementing the synchronizations and communications associated to the data dependences on each case, is cumbersome and error-prone. It is increasingly interesting to support these applications in high-level parallel programming languages or parallelizing compilers. There is a lack of support for this kind of applications in most automatic parallel code generators for distributed memory.

In this chapter we present a new technique based on the same ideas of the one presented in the previous chapter, but devised to calculate the specific communication patterns derived from data-parallel codes with or without periodic boundary conditions on the uniform affine data-access expressions. It addresses the fifth goal of the Thesis:

Extend the technique to automatically calculate aggregated communications in applications with periodic domains.

9.1 Introduction

Many real-world applications feature data accesses on periodic domains. For example, physical phenomena can be modelled using spherical grids with periodic boundary conditions [112], using a stencil program over a periodic domain. Manually implementing the synchronizations and communications associated to the data dependences for each specific application, is cumbersome and error-prone. Thus, it is increasingly interesting to support these applications in high-level parallel programming languages or parallelizing compilers.

In this chapter we present a technique that, automatically calculates at runtime the coarse-grained communication patterns that are needed for a correct and efficient execution of SPMD (Single Program Multiple Data) programs derived from data-parallel codes with periodic uniform affine expressions in data accesses. It inherits the nice features of the previous technique, making transparent to the programmer the management of aggregated communications for the chosen data partition. It also moves to runtime part of the compile-time analysis needed to generate the communication code. Thus, it also produces programs more adaptable to different execution environments, allowing, for example, the use of different tile sizes at the same hierarchical level, a good approach for clusters that include machines with different architectures [87].

To show the applicability of our approach, we also develop the technique in the Trasgo parallel programming system proposed in [54]. We test six benchmarks with periodic access expressions: An illustrative example based on the rotate routine of the STL library [124], the periodic versions of Heat 1-D, 2-D and 3-D applications [23], a matrix multiplication program using Cannon's algorithm [30] and a multi-grid V-cycle 3D-stencil application, the NAS MG benchmark [11]. Our experimental results, comparing pure MPI reference codes and the programs automatically generated, in both distributed- and shared-memory environments, show that the use of our approach can automatically obtain efficient codes while reducing the development effort. For example, the use of our solution leads to a reduction of 44.42% in the number of code lines and a reduction of 65.71% in the McCabe Cyclomatic Complexity for the NAS MG Benchmark.

9.2 Related work targeting problems with periodic domains

Applying the tiling technique to a set of indexes enables a medium-grained parallelization. Some approaches have been presented to solve the problem of tiling with periodic boundary conditions. The closest method to our approach is a cutting-and-pasting technique. In this technique, the dependencies which are affected by the periodic conditions are broken and displaced. It is similar to a circular loop skewing. This approach needs a computation of the transitive closure of dependencies to determine the set of iterations which another tile depends on. Some libraries such as ISL [135] are capable, for some problems, of computing

```
** Sequential rotate algorithm
Inputs:   size: Vector size
          <type> M[size]: Vector with initial values
          f: function to compute each element
          rot: Amount of positions to rotate the elements


Outputs:  <type> M2[size]: Vector with result values

   1. <type> M2[size];
   2. For i = 0 to size-1
         M2[i] = f(M[(i+rot) mod size])
```

Figure 9.1: Sequential algorithm for the illustrative example assuming a positive value of *rot*.

at compile time the transitive closure of dependencies efficiently. However, the transitive closure computation istypically a very hard problem for solutions that work at compile time. Another method to tile and optimize time-iterated computations over periodic domains was presented in [23]. This technique first splits the iteration domain, cutting close to the mid-point what their authors call *long dependences*. After this cut, they apply a separate affine transformation on each half of the space. All these approaches target shared-memory systems. Programming for distributed-memory systems is more challenging due to the management of the distributed data structures. Moreover, communications among processes should be devised in terms of, for example, message-passing operations, taking into account all the potential combinations of proper data partitions or matrix sizes. Our technique makes transparent all these issues.

9.3 Illustrative example

This section presents an illustrative application to show an example of how a parametrizable algorithm can be tackled with our proposed solution. We selected an example based on the *rotate* routine of the STL library [124]. Refer to the sequential algorithm in Fig. 9.1. The routine applies a rotation to the vector elements while applying a function $f()$ on each one of them. The routine receives an integer parameter *size*, a vector of elements (with the size indicated by the previous parameter), the function to apply yo the elements *f()*, and another integer parameter *rot*. The parameter rot is used in the access expressions to select, at runtime, the amount of positions that the elements in the vector should be shifted.

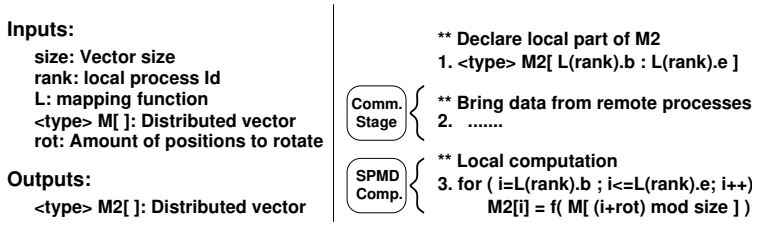

Figure 9.2: Parallel algorithm for the illustrative example in a SPMD model assuming a previously distributed input array. Boxes indicate the logical steps in SPMD computations.

A distributed parallel version of this algorithm is shown in Fig. 9.2. For simplicity, we assume that the function *f()* is fixed and not received as parameter. In this case, the input parameters are: (1) The size of the original input vector, $size$; (2) the local process identifier, $myRank$; (3) a mapping function L that receives a process identifier and returns the set of indexes, in the range [0:*size*-1], assigned to that process (b and e will indicate the limits of the set for this process) ; (4) An input vector M that was previously distributed among the different processes using the mapping function L; (5) The integer parameter rot that is assumed to be positive in this example.

First, the local part of the distributed output vector $M2$ is declared, also using the mapping function L. Thus, each process stores the same part of both arrays, M and $M2$. After that, each process participates in the rotation of the whole vector elements, and applies the function $f()$ on its part (SPMD block). When a process applies the access expression $(i + rot)\ mod\ size$ to its assigned index domain, some element indexes access to data in remote partitions and it is possible that part of the resulted indexes are out of the assigned domain. Notice that the mod operator makes these indexes access to elements in non-neighbor processes. Hence, we insert a communication phase before the SPMD block to ensure that every process has the necessary data to compute $f()$. After the communication and computation, processes will be able to update its part of the $M2$ output vector. Our technique calculates automatically these necessary communications.

We illustrate the communication calculation for this example in Fig. 9.3. In the left of the figure, we see in Stage 1 the set of indexes assigned by the mapping function L to each process. In Stage 2, we see the set of indexes resulted by applying the expression $(i + rot)$ to the indexes assigned to the processes 2 and 4. In our example, the domain accessed by process 2 overlaps with the domain owned by process 3. Thus, in order to compute, process 2 needs to receive those data from process 3. For process 4 we obtain two sets of indexes: (a) A set located into the original array domain (it will be named s°), and (b) a set located outside of the original array domain. This set is represented in yellow and it will be named s^{+}. Notice that s^{+} is a set of indexes higher than the end of the array domain, due to the

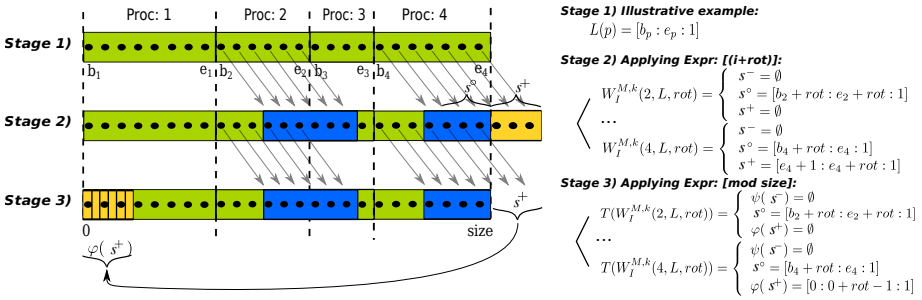

Figure 9.3: Communication structures calculation. Using the read data-access expression inside the parallel computation of the illustrative example to build the functions that calculate the working input indexes set (W_I), for the data structure M, for the processes 2 and 4. This example uses contiguous rectangular shapes, an irregular data partition policy, and depicts the particular case of $rot = 3$, for a domain with $|M_0| = 25$.

positive value of rot in the example. If the value of rot had been negative, we would have had a set of indexes lower than the start of the array domain, that we name s^-.

The application of the periodic boundary condition ($mod\ size$) over the set of indexes resulted of applying $i + rot$ in process 4, transform them to the set of indexes shown in Stage 3. The yellow region on Stage 2 corresponds to the yellow stripped region that belongs to process 1 at Stage 3. We will need an extra communication to receive the stripped region from process 1 at process 4.

9.4 Aggregated-communication model

In this section, we present a generic model that calculates at runtime the necessary communication patterns to compute communications for SPMD blocks, that contain uniform affine access expressions on the indexes of the parallel loops (SPMD blocks) with optional periodic boundary conditions. In this work we assume an *owner-computes* paradigm, with the result of a runtime mapping function indicating the ownership. We describe the model to calculate communication patterns to move data from owner processes to the processes where these data are accessed to execute computation. The same model can be applied to move computed data to destination positions, in the case of write accesses which are also generic expressions that transform the domain indexes.

In this section, first, we present the notation used. After that, a simplified one-dimensional overview of the communication calculation is presented. Finally, the generic multi-dimensional model is described.

9.4.1 Definitions

In this section, we define the terms used during the model description. We use lower-case characters (as s or s') as generic elements of a given set. We use the dot operator to refer to the fields of a tuple (for example, $s.b$ represents the field b of the tuple s).

- **Number of elements** (N): number of elements in an array M.
- **Number of dimensions** (n): number of dimensions of an array M.
- **Cardinality** ($|M_x|$): number of indexes of the array M in the x-th dimension.
- **Signature** ($S\langle b, e, z\rangle$): is a triplet of three integer numbers *begin* (b), *end* (e), and *stride* (z). A signature defines a subset of integer one-dimensional indexes from the *begin* to the *end*, using the *stride* as step. We will use the classical Fortran90 notation $[b : e : z]$ for simplicity in our discussion.
- **Hyperrectangle domain** ($d \in D_n$): An n-dimensional domain formed by the Cartesian product of n signatures (a hyperrectangle).
- **Domain** ($d \in D_n^*$): A set of n-dimensional indexes. Any set of n-dimensional indexes can be represented as the union of hyperrectangle domains, which are a particular case of domains ($D_n \subset D_n^*$). In a similar way as the Kleene star or closure operator [46] denotes all the possible combinations of a set of strings using the concatenation operator, we use D_n^* to denote the set of all possible domains of n-dimensional indexes, built as the union of hyperrectangle domains.
- **Computation indexes** ($\overrightarrow{i}$): The set of indexes where a parallel computation will be performed.
- **Affine expression** ($\rho_x(\overrightarrow{i})$): In our technique we consider affine expressions on the x-th dimension with the form:

$$\rho_x(\overrightarrow{i}) = \alpha_0 \times i_0 + ... + \alpha_{n-1} \times i_{n-1} + \beta$$

  where the coefficients $\alpha_{0..n-1}, \beta$ are invariant in the body of the SPMD block. We can also apply an affine expression to a whole set of indexes described by a signature ($\rho_x(s)$).
- **Periodic expression** ($cyc(\rho_x(\overrightarrow{i}))$): It denotes periodic boundary conditions on the result of an affine expression.
- **Mapping function** ($L(p)$): It is a function that receives the index of a process and returns the domain representing the set of indexes mapped to that process.

- **Access expression** ($M[cyc(\rho_0(\overrightarrow{i}))]...[cyc(\rho_{n-1}(\overrightarrow{i}))]$): An expression in the code that accesses the data in a data structure M. A periodic boundary condition is applied on each dimension of the data structure.
- **Set of affine expressions** ($\phi(\overrightarrow{i})$): The set of affine expressions (one for each dimension) of a single access expression.

9.4.2 Model for calculating communication patterns in 1-D applications

In this section we present an overview of the proposed technique to calculate the communication patterns for only one-dimensional index domains. Recall the parallel algorithm presented in Fig. 9.1, that performs the rotation of the elements of a vector also applying a function $f()$. A communication phase is needed before the SPMD block to reallocate some data across processes, ensuring that each process has the necessary data to compute.

The following analysis is done independently for each data structure, and for each SPMD block. The *Input Working Set of Indexes* ($W_I^{A,k}(p, L, \overrightarrow{\delta})$) is a function built based on the access expressions on a SPMD block. It returns the set of indexes of the data structure A, read by a given processor p, during the k-th SPMD block in the code (remember Stage 2 of Fig. 9.3). The parameters of the function are: The processor identifier p, a mapping function L that returns the set of indexes mapped to any processor, and $\overrightarrow{\delta}$, the values of the symbolic parameters that appear in the expressions. The function applies at runtime the affine expressions (ϕ) found in read accesses to A inside the k-th SPMD block, one by one, to the indexes set returned by the mapping function ($L(p)$). See a calculation example in Stage 2 of Fig. 9.3.

For the one-dimensional case, we obtain a set of indexes that can be represented by a set of signatures. These signatures can be classified as:

- Set of signatures whose indexes are lower than the begin of the original array domain (S_0^-).
- Set of signatures whose indexes are included in the original domain (S_0°).
- Set of signatures whose indexes are higher than the end of the original array domain (S_0^+).

We generate the code of the functions that compute the set of indexes read per each SPMD block, and each data structure ($W_I^{A,k}(p, L, \overrightarrow{\delta})$). These functions return a set of signatures representing the set of indexes read. With these functions, a process can calculate the input working set of the distributed data structure, mapped to any process, once the parametric values are known. These functions simply apply at runtime the access expressions to the index-space limits at runtime to calculate the working-sets (see the implementation of these functions in Sect. 9.5).

Once we have the set of signatures that result of applying the affine expressions, we can apply the periodic boundary conditions, where it is required. This relocates the indexes that after the application of the affine expression are out of the original array domain, into the domain. See Stage 3 in Fig. 9.3. We define two functions to apply the periodic conditions to signatures. Remember that $|M_0|$ is the number of elements of M in the first dimension.

- For signatures in S^- we define $\psi : S \to S$ where

$$\psi(s) = s' : \begin{cases} s'.b = -((-s.b)\ mod\ |M_0|) + |M_0|, \\ s'.e = -((-s.e)\ mod\ |M_0|) + |M_0|, \\ s'.z = s.z \end{cases} \tag{9.1}$$

- For signatures in S^+ we define $\varphi : S \to S$ where

$$\varphi(s) = s' : \begin{cases} s'.b = s.b\ mod\ |M_0|, \\ s'.e = s.e\ mod\ |M_0|, \\ s'.z = s.z \end{cases} \tag{9.2}$$

A function $T : D_n^* \to D_n^*$ transforms the working input set by applying the two previous functions to relocate all the indexes back into the original array domain, as it was showed in Stage 3 of Fig. 9.3. We can express the transformation with the following function:

$$T(W_I^{A,k}(p, L, \overrightarrow{\delta})) = \begin{cases} if\ s \in S^-; s' = \psi(s), \\ if\ s \in S^\circ; s' = s, \\ if\ s \in S^+; s' = \varphi(s) \end{cases} \tag{9.3}$$

An example of the application of $T(W_I^{A,k}(p, L, \overrightarrow{\delta})$ for the illustrative example can be seen also in Stage 3 on the right of Fig. 9.3.

Algorithm 3 uses a simplified one-dimensional model to calculate the data to be received at any process. The output is a set of communication tuples (C_R). A comm-tuple $\langle p, D^* \rangle$ associates the index of the remote process p, with the set of indexes D^* of the structure whose data values should be communicated. For each data structure, the local process, named $myRank$, calculates the exact data to be received from a remote process p. In order to do that, local process intersects $L(p)$, which is the domain assigned to that remote process by the mapping function, with the data positions needed in read accesses, which are represented by the transformed input set at the local process, $T(W_I^{A,k}(myRank, L, \overrightarrow{\delta}))$. If the intersection is not empty, a receive communication should be performed. The data have to be received in the positions accessed by the local process, so the applied boundary conditions are reverted. The data to be sent to process p can be calculated by the opposite intersection: The assigned domain to the local process ($L(myRank)$), with the transformed

Algorithm 3: Model to calculate the receive communication pattern for a SPMD block, for a given data structure A.

Input:
P: Number of processes;
$myRank$: Local process id;
$L()$: Mapping function;
$\overrightarrow{\delta}$: Symbolic parameters;
$|A_0|$: Cardinality of the 1-D array;
$W_I^{A,k}()$: Function to compute the working input set;
$\psi(),\ \varphi()$: Periodic transforming functions
Output: C_R : Set of comm-tuples that indicates data to be received
$C_R \leftarrow \emptyset$
for *p: 1 to P* **do**
  $lp \leftarrow L(p)$
  for *all* $s \in T(W_I^{A,k}(myRank, L, \overrightarrow{\delta}))$ **do**
    $s' \leftarrow s \cap lp.S_0$
    if $s' \neq \emptyset$ **then**
      $tmp_0 \leftarrow \emptyset$
      if $p \neq$ *myRank and* $s \in S^{\circ}$ **then**
        $tmp_0 \leftarrow s'$
      end
      if $s \in \psi(S^{-})$ **then**
        $tmp_0 \leftarrow s' - |A_0|$
      end
      if $s \in \varphi(S^{+})$ **then**
        $tmp_0 \leftarrow s' + |A_0|$
      end
      $C_R \leftarrow C_R \cup \langle p, \langle tmp_0 \rangle \rangle$
    end
  end
end

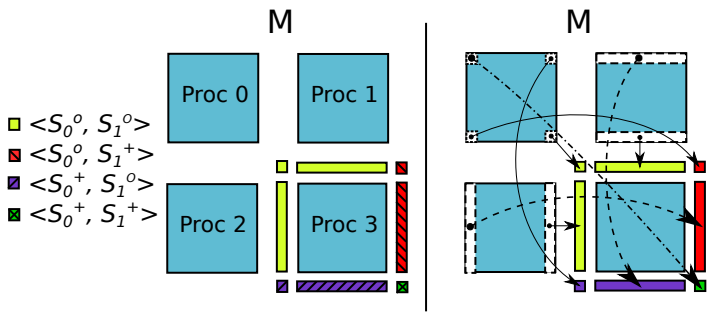

Figure 9.4: Stencil-2D application: Input matrix has been divided among four processes. Left: Set of signatures accessed by process 3 and their classification. Right: Communications needed to enable the computation at process 3, represented by arrows.

input sets at the remote process ($T(W_I^{A,k}(p, L, \overrightarrow{\delta}))$). Empty intersections indicate that no send or no receive operation is needed for that particular process p. In the illustrative example of Sect. 9.3, the groups S^- is empty for any process p, due to rot being positive, and simply added to the parallel loop index in the expression. See the details in Fig. 9.3.

9.4.3 Multi-dimensional model

As we have seen in the previous section, for the one-dimensional case, we have three groups of signatures, depending on the access expressions (S^-, S°, S^+). However, in a general case with n dimensions, the number of classes for the transformed domains, before applying boundary conditions, is 3^n. For example, let $A[cyc(\rho_0(\overrightarrow{i}))][cyc(\rho_1(\overrightarrow{i}))]$ be a two-dimensional periodic access expression. The input working set is calculated similarly as in the 1-dimensional case, simply applying the access expressions to the index-space limits at runtime for every dimension. As we use hyperrectangular tiles, no over-approximation is performed, so the calculated input working set only contains the index space obtained by applying the access expressions on the different dimensions.

The domains that compose a 2-dimensional input working set can be classified in 3^2 possible groups of domains. The classification groups are defined as the combination of the S^-, S°, and S^+ sets of each dimension:

$$\langle S_0^\circ, S_1^\circ\rangle, \langle S_0^\circ, S_1^-\rangle, \langle S_0^\circ, S_1^+\rangle, \langle S_0^-, S_1^\circ\rangle, \langle S_0^-, S_1^-\rangle,$$
$$\langle S_0^-, S_1^+\rangle, \langle S_0^+, S_1^\circ\rangle, \langle S_0^+, S_1^-\rangle, \langle S_0^+, S_1^+\rangle$$

Figure 9.4 shows an example based on a Stencil-2D application with periodic boundary conditions. For simplicity in this figure, the matrix M has been divided in four equal parts that correspond to 4 processes. Thus, each process computes a part of the matrix. In this example each process needs data of its neighbors for computing. On the left of the figure, we

```
1  /* Function to rotate one element */
2  void rotateElem(in double m, out double m2 ) {  m2=f(m); }
3
4  /* Rotate: Parallel function */
5  coordination void rotate( in tile double M[], in int rot,
6                            in int size, in Map L, out tile double M2[]) {
7      ArrayMap( M2, L );
8      parallel(i in [0 :size-1]){
9          rotateElem(M[cyc(i+rot)], M2[i]);
10     }
11 }
```

Figure 9.5: Trasgo input code for the illustrative example.

show the domains accessed by process 3, classifying also the domains in their corresponding groups (e.g $\langle S_0^\circ, S_1^\circ \rangle, \langle S_0^\circ, S_1^+ \rangle$). The groups of domains not represented in the figure are empty. On the right of the figure, we represent using arrows the corresponding receive communication operations that should be performed on each iteration by process 3, and which are automatically calculated by our proposal.

The multidimensional model uses an extension of the Alg. 3. In this case, the computation of the transformed input working set ($T(W_I^{A,k}(p, L, \overrightarrow{\delta}))$), and the reversion of the resulting domains are done by applying the transforming functions (ψ and φ) independently on each dimension.

9.5 Implementation on a parallel programming framework

We implement this proposal also in the Trasgo system. Figure 9.5 shows a simplified example of how to program the illustrative example presented in Sect. 9.3 for Trasgo. First, we define the sequential function to apply to each data element, specifying the output or input role of the parameters (line 2). The parallel function is defined using the *coordination* modifier, and also specifying the role of the parameters (lines 5 to 6). In its body, the function *ArrayMap()* allocates the distributed array $M2$ in terms of the results of the mapping policy L, that is also a parameter (line 7). After the distribution, the code updates each element of $M2$ in parallel invoking, inside the *parallel* statement, the sequential function previously defined (lines 8-9).

We implemented the proposed technique on top of the Hitmap library (remind Sect 8.3.3). Mapping functions are represented in Hitmap as *HitLayout* objects; The signatures by *HitSig* objects; The hyperrectangular domains by *HitShape* objects; And the sets of domains by

```
/** Calculate W_I for M in SPMD */
HitDomain calculateWI_1_M( HitRank p, HitLayout lay,
                           HitTile Tile1, int rot){
   // L(p)
   HitShape remote = hitLayOtherShape( lay, p );
   // 1*begin+rot, 1*end+rot, 1*stride
   HitDomain inWS = hitShapeAffine1( remote, 1,+rot);
   //Apply boundary conditions
   hitApplyBoundary(inWS, lay, Tile1);
   return inWS;
}
```

Figure 9.6: Excerpt of the generated function that applies the input-code affine access expressions and periodic conditions to compute $T(W_I^{A,k}(p, L, rot))$ for the illustrative example.

HitDomain objects. The handlers containing pointers to the actual data structures are *HitTile* objects.

We implemented the multidimensional model of the proposed technique on Trasgo, using the Hitmap features, for efficient domain set operations on hyperrectangular shape structures such as intersection $\cap$, union $\cup$, and subtraction $\setminus$.

Figures 9.6 to 9.7 present an example of the functions automatically generated following the proposed technique to calculate the receive communication pattern (C_R) for the illustrative example, and the lines to be inserted in the main code to invoke these functions.

The function named *calculateWI_1_M* (see Fig. 9.6) is generated at compile time by Trasgo. It uses three Hitmap functions: (1) *hitLayOtherShape* returns the domain assigned to a given remote process; (2) *hitShapeAffine1* applies an uniform affine access expressions of the form $(\alpha_0 \times i_0\beta)$ to the index-space limits of the first dimension at runtime. (It performs the computation of Stage 2 of Fig. 9.3); (3) *hitApplyBoundary* applies the boundary conditions, and returns a set of signature domains bounded to the original array domain. It performs the transformation shown at Stage 3 of Fig. 9.3.

We implement the algorithm presented in section 9.4 into the Hitmap library as a function. It receives as parameters the mapping function used to part the data structure (a HitLayout object), and the generated functions for the specific piece of code needed to calculate the transformed input working set ($T(W_I^{A,k}(p, L, \overrightarrow{\delta}))$). The result of the new function `calcComms`, is a HitPattern object where the calculated comm-tuples have been inserted. Figure 9.7 shows how both the communication calculation and the execution functions are called in the target main program. A similar piece of code is automatically inserted before the execution of each SPMD block.

```
/* Building comm. pattern */
HitPattern _TT_comA = calcComms(M, calculateWI_1_M, M.Layout, rot);
/* Communication, execute pattern */
hit_patternDo( _TT_comA );
```

Figure 9.7: Calling both the communication calculation and execution functions in the target program of the illustrative example.

9.6 Discussion: Analyzing the technique

Communication optimality: For a given input working set, our technique calculates the corresponding exact data communication. In the current prototype framework, the construction of the exact input working sets is limited for some kind of programs.

In our implementation, the domains used to represent the indexes accessed in a given piece of code should be represented as hyperrectangles (signature domains). The representation of any other shape should be done as a union of signature domains. The number of signature domains needed is directly related with the runtime complexity of applying our technique. We developed a function that, after a union of domains, eliminates the redundant elements in the communication object. It is represented in the Alg. 3 by the $\cup$ symbol. This function will be applied at runtime before the communication execution. The asymptotic complexity of this function is dependent on the number of signature domains stored in the communication object, whose data will be communicated. Using this function we avoid redundant communication. This is one of the main advantages of our technique with respect to the previous work. However, depending on the program, the complexity of this function can penalize the performance of the application in some cases. A future work will determine the best option between applying the function to eliminate the redundant communications, or communicating extra data on each case.

Future work also includes the integration of polyhedral frameworks, which use more sophisticated representations [132], to extend the application range of our implementation.

Scalability on computational units: Analyzing Alg. 3 we observe that the time spent by the communication calculation grows linearly with the number of processes. This has been verified in the experimental study in Sect. 9.7.4. This trend also appears in other previous distributed-memory approaches used to derive communications code (remind Chap. 6). For scalability in target platforms with high orders of magnitude of processing elements, these techniques should be combined with other ones, such as hierarchical groups of processes, or the detection and application of specific techniques for application patterns, as we shown in the experimental study of Sect 8.5.4.

Table 9.1: Input data sizes (N) and time loop iterations (T), for benchmarks in the experimental studies.

Study 1 Bench.	Sizes (N)
Rotate	N =$3 * 10^7$, rot=2
Cannon	N=7680 x 7680
MG	Class D

Study 2 Bench.	Sizes (N), iterations (T)
Heat-1d	N = 2000000, T = 6000
Heat-2d	N = 8000 x 8000, T = 500
Heat-3d	N = 500x500x500, T = 100

9.7 Experimental study

We performed an experimental study to validate our approach, and to verify the efficiency of the resulting codes, studying the potential overheads introduced by our runtime calculation. The section is divided into: (1) the experiment design details, (2) a study of several study cases, comparing our proposal with optimized MPI reference programs in terms of performance and code complexity; and (3) a breakdown of the performance measures of our codes in computation, communication calculation, and communication execution times.

9.7.1 Design and setup of the experimental study

The experiments were executed in two platforms. The first one (CETA) is a hybrid cluster that belongs to CETA-CIEMAT[1] and the Spanish government. The cluster nodes are connected by Infiniband, and they have two Intel Xeon 5520 CPUs at 2.27 GHz, with 4 cores each. Using 8 nodes of the cluster, we exploit up to 64 computational units. The second platform is a pure shared-memory machine (Heracles), a Dell PowerEdge R815 server, improved with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, and 64 cores in total. We compiled the codes with the GCC v6.2 compiler, using the optimization flag *-O3*. As MPI implementation, we use *mpich3* v3.1.3, with device $ch4$, that also improves communication performance on shared memory.

We use static mapping, launching one MPI process for each processing element. The main contribution of this work is the technique to calculate automatically the data communications needed when an application with periodic array accesses is executed on a distributed-memory system. Thus, in this chapter, for a fair comparison, we use only MPI processes without exploiting threads programming. We focus on studying only the potential overheads introduced by this calculation, and the benefits obtained due to the aggregated communications. The foundation of how OpenMP threads can be efficiently composed inside MPI processes when using Hitmap was presented in [95].

[1] Extremadura Research Center for Advanced Technologies, Spain.

Table 9.2: Study 1: Performance (in seconds) for the illustrative example, Cannon's algorithm, and the MG real-world application. Comparison of MPI references and Trasgo generated codes. Notice that Cannon's algorithm requires a number of processes with a perfect square root.

	Rotate		Cannon		MG	
Machine	**MPI**	**Trasgo**	**MPI**	**Trasgo**	**MPI**	**Trasgo**
Heracles-4	1.00	1.26	148.42	146.13	512.05	519.05
Heracles-8	0.66	0.84	--	--	364.47	367.80
Heracles-16	0.35	0.43	42.43	41.69	208.59	214.40
Heracles-32	0.24	0.32	--	--	135.44	138.46
Heracles-64	0.09	0.11	11.95	10.66	105.28	100.16
CETA-4	0.94	1.07	101.16	106.20	--	--
CETA-8	0.50	0.57	--	--	--	--
CETA-16	0.27	0.31	29.89	28.60	210.92	231.83
CETA-32	0.14	0.16	--	--	194.14	229.99
CETA-64	0.08	0.08	10.68	12.80	151.66	197.64

We executed the programs in CETA and Heracles with number of processes $P = 1$, 4, 8, 16, 32, and 64. We performed ten executions per each test, taking the average time. We selected big enough input data sizes to produce a minimum computational load that remains significant when the computation is distributed across 64 computational units. The input data sizes (N) and time loop iterations (T), for the different benchmarks in the experimental studies are presented in Tab. 9.1. All the codes use a data partition policy that splits the input data structure in as many 1D, 2D, or 3D blocks as number of processes.

9.7.2 Study 1: Performance comparison with MPI reference codes

In this section we present a performance study using: (1) The simple illustrative example presented in Sect. 9.3; (2) The well-known Cannon's distributed-memory parallel algorithm for matrix multiplication [30], which is specially devised for distributed-memory systems in order to minimize the memory footprint; and (3) The real-world application MG of the NAS Benchamrks, implementing a multi-grid v-cycle method for a 3D-stencil computation [11].

In this study we perform an end-to-end time measure including all program stages where Trasgo could introduce overheads. Table 9.2 shows the performance obtained when we compare Trasgo generated codes with reference MPI versions. Our programs for the illustrative example and Cannon's algorithm scale similarly to the optimized MPI codes.

The NAS MG benchmark requires further discussion. The distribution of the data structures is key to execute this computationally-intensive application. The MG data structures for the D input class (defined by NAS benchmarks) need to be distributed at least among 16 processes in order to fit in the local memories of the nodes of our distributed-memory machine, CETA. Thus, we only present the results for 16, 32 and 64 processes on Tab. 9.2. The MPI reference code of the NAS MG benchmark contains a manual optimization to

Table 9.3: Comparison of development effort measures for three case studies.

		KDSI	McCabe's C.C.	Halstead D.E.
	Trasgo	24	6	74K
Rotate	C+MPI	62	21	1 890K
	Reduction	61.29%	71.43%	96.08%
	Trasgo	57	4	19K
Cannon's MM	C+MPI	175	4	122K
	Reduction	67.43%	0.00%	84.43%
	Trasgo	772	72	19 477K
NAS MG	C+MPI	1389	210	29 568K
	Reduction	44.42%	65.71%	34.13%

communicate data across different levels of the v-cycle. This optimization cannot be directly derived from the access-expression analysis of the SPMD blocks that traverse the multi-grid during the v-cycle. Our implementation issues an extra communication phase that this optimization eliminates, incurring thus in a performance loss up to 30% in our measures.

In summary, our technique allows the automatic calculation of communication stages for codes with uniform affine expressions with periodic boundary conditions at runtime efficiently. As we stated above for MG, the drawback of this kind of approaches is that these automatic calculations do not generate certain communication optimizations across SPMD blocks that could positively impact performance. An open question is whether these particular optimizations could be automatically applied after the use of this kind of techniques.

9.7.3 Study 2: Ease of programming

Our technique avoids to the programmer the management of the communication and/or data partition codes. This leads to a reduction on the parallel programming complexity. Table 9.3 shows, for our study cases, several complexity and development effort metrics, including KDSI metric used in the COCOMO model [19] (number of lines), McCabe's cyclomatic complexity [86], and Halstead development effort [63]. These metrics are used to compare the potential programming effort needed when using the different alternatives considered. We observe that the development effort needed is highly reduced when using our approach. As can be seen in Tab. 9.3, the reductions for the different metrics used range from 44% to 96% in all cases, except in the McCabe complexity for the Cannon's matrix multiplication, where this measure is extremely low in both codes.

9.7.4 Study 3: Relative cost of calculating communications

Our technique to calculate the communication patterns is performed at runtime. In this section we show an experimental study where we focus on the cost of our calculation and

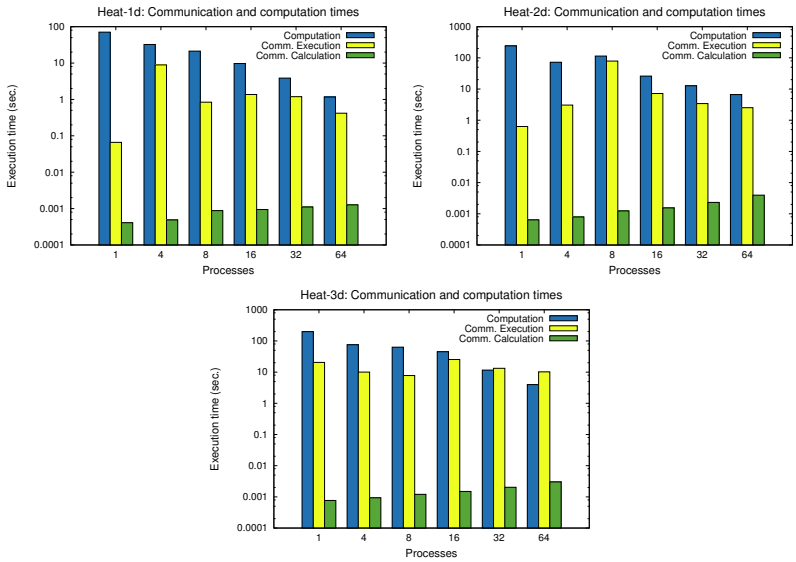

Figure 9.8: Computation, communication calculation, and communication execution times in seconds for the Heat examples on the distributed-memory machine (log scale), using the problem sizes of Tab. 9.1.

synchronization times with respect to the main computation times. The experiment was executed in two different architectures, the distributed- and the shared-memory machines, *CETA* and *Heracles*.

Figure 9.8 and 9.9 show the measures of the computation and the communication times, also separating in communication calculation and communication execution times. We show results for the periodic versions of the Heat-1d, Heat-2d and, Heat-3d benchmarks [23], for different number of MPI processes launched (notice the logarithmic scale in the plots). We see that the computation time decreases when the number of processes increases, except in one situation (Heat-2D with 8 processes in the distributed-memory system). This phenomenon is not related to the data communication that is the focus of this study. We also observe that the time spent by our technique in the runtime calculation of the communication patterns increases with the number of processes, as expected. However, these times can be consider negligible, as they are several orders of magnitude smaller than the computation and the communication execution times.

In summary, our technique automatically and efficiently calculates at runtime the communication patterns needed in a distributed-memory parallel program with periodic access

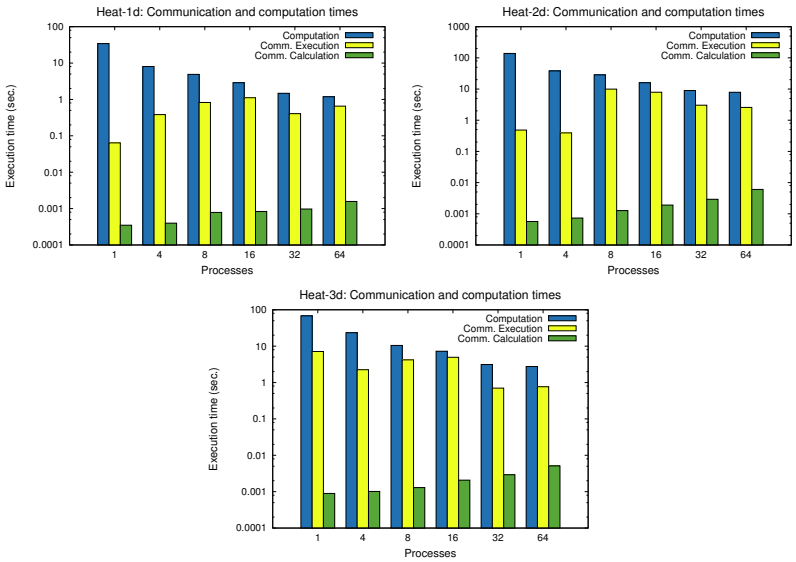

Figure 9.9: Computation, communication calculation, and communication execution times in seconds for the Heat examples on the shared-memory machine (log scale), using the problem sizes of Tab. 9.1.

expressions, allowing the selection at runtime of the partition policies, and the choice of the proper tile sizes for the actual execution platforms.

9.8 Summary

This chapter describes a technique that calculates at runtime exact aggregated coarse-grained distributed-memory communications, for algorithms with uniform affine expressions with periodic boundary conditions. It is based on: (1) calculating at runtime different footprints through cutting-and-pasting methods in terms of the mapping functions chosen and, (2) intersecting at runtime the remote and local footprints. Performance results for six cases of study, including a real-world benchmark, indicate that using our technique, we obtain similar efficiency to MPI codes, while the development effort is reduced. Future work includes also the applicability of the proposed technique in current polyhedral model frameworks.

CHAPTER 10

Operators for data redistribution

As we have seen in previous chapters several tasks related to the data management in distributed-memory spaces can be automatized. Nevertheless, there are still many cumbersome optimizations and techniques that should be manually tackled by the programmer to obtain really efficient parallel programs for many application structures. One of the optimizations most difficult, error-prone, and useful at the same time, for distributed-memory systems, is the data redistribution. Data redistributions allow the improvement of the performance of algorithms which operate on changing or partial domains along the program execution, by creating a balanced workload among the active processes on each stage.

In this chapter we address the last goal:

Propose an abstraction to simplify data redistributions.

In order to do that, we present four operators to redistribute selected data on distributed-memory systems in an efficient and simple way, making the management of the data partition, relocation, and data movement transparent to the programmer.

10.1 Introduction

The message-passing paradigm (implemented for example by MPI libraries) has been shown to be a programming method for distributed-memory systems that leads to highly efficient programs in terms of performance. However, the programmer still has to deal with many decisions not related with the parallel algorithms, but with implementation issues, such as decisions about partition and locality vs. synchronization/communication costs, scheduling details, etc.

One of these optimizations, for static-scheduling approaches on distributed-memory systems, most useful is the data redistribution. Data redistributions allow the improvement of the performance, by creating a balanced workload among the active processes. However, its programming is complicated and error-prone. The programmer has to take into account many details such as the partition policy used initially to distribute the data among the processes, the partition policy desired after the redistribution, and the number of active processes on each stage. Moreover, recursive partitions and data dependent selections complicate the decisions that should be taken at runtime.

In this chapter we present four operators to efficiently redistribute selected data, making the data partition, relocation, and data movement transparent to the programmer. These operators are applied on an array already divided across distributed processes, and the result is another array containing all or part of the original array elements relocated across the available processes. These operators can be freely combined, even in a recursive algorithm. We implement the four operators as an extension of Hitmap [55].

10.2 Motivating example

This section presents a motivating example to show the advantages of using data redistributions. We choose an example where redistributing data is needed to create load-balance and improve efficiency. However, programming this data redistribution in a plain message-passing model is difficult. A simplified parallel algorithm of the motivating example, using a message-passing approach, is presented on the left of Fig. 10.1. It initializes a vector and then, it updates in parallel a number of elements (a tenth of the original array size) centred around an arbitrary chosen element. This algorithm is used in real applications, as simulations of computational fluid dynamics. It recalculates only values in a significant neighborhood surrounding a point of the input array that experiments a sudden abrupt change, without recalculating the values of the whole array. The input parameters are: (1) An integer $Size$ that determines the size of the input array. (2) An integer pos that indicates the position of the main element whose neighbors will be updated. (3) An integer id that represents the identifier of the local process. (4) An integer P that contains the number of processes, and (5) an array M of $Size$ elements already distributed among the P processes. $A\langle type\rangle$ denotes a

```
** Distributed algorithm in
** message-passing
Inputs:
 int Size: Vector size
 int pos: main position to update
 int id: Local process identifier
 int P: Number of processes
 A<type> M[]: Distributed Vector
Outputs:
 A<type> M_out[]: Distributed Vector

1. ** Each process initializes a
    ** part of the input vector
myRange.begin= id * Size/P;
myRange.end=myRange.begin+Size/P-1;
for ( i=myRange.begin;
      i<=myRange.end; i++)
     M[i] = init();

2. ** Calculate the range of
   **  the neighbors
Nelemt=Size/10;
first= (pos-Nelemt/2);
last= (pos+Nelemt/2);

3. ** Calculate the redistribution
myRange2.begin= id*Nelemt/P+first;
myRange2.end= myRange2.begin-1+
              Nelemt/P;

for(id_p=0; id_p< P ;id_p++){
 range.begin= id_p * Size/P;
 range.end=range.begin + Size/P-1;
 range2.begin= id_p * sel/P;
 range2.end=range2.begin + sel/P-1;
 Range send_p =  intersect(
     <myRange.begin,myRange.end>,
     <range2.begin,range2.end>);

 Send(M, send_p, id_p);
 Range recv_p =  intersect(
     <myRange2.begin,myRange2.end>,
     <range.begin, range.end>);
 Recv(M_out, recv_p, id_p);
}

4. ** Update the elements
for ( i=myRange2.begin;
      i<=myRange2.end; i++)
      M_out[i]=Compute(...)
```

```
** Distributed algorithm
** using proposed operators
Inputs:
 int Size: Vector size
 int pos: main position to update
 int id: Local process identifier
 int P: Number of processes
 Map L: Mapping function
 A<type> M[]: Distributed Vector

Outputs:
 A<type> M_out[]: Distributed
                  Output Vector

1. ** Each process initializes
   ** a part of the input vector
for ( i=L(id,Size,P).begin;
      i<=L(id,Size,P).end; i++)
      M[i]=init();

2. ** Calculate the range of
   **  the neighbors
Nelemt=Size/10;
first= (pos-Nelemt/2);
last= (pos+Nelemt/2);

3. ** Redistribution
M_out=ArrayRemapRange
          (M,<first,last>,L)

4. ** Update "Nelemt" elements
for ( i=L(id,Nelemt,P).begin;
      i<=L(id,Nelemt,P).end;
      i++ )
      M_out[i]=Compute(...)
```

Figure 10.1: Motivating example algorithms using two different approaches: Reference algorithm in message-passing style (left); and programmed using the proposed operators in a single-program-multiple-data model (right).

distributed array. For simplicity, in this example, we assume that $Size$ is divisible by P, and the data were originally mapped assigning to each process $Size/P$ elements with contiguous indexes.

The output will be the distributed array M_out containing the updated elements.

In the first stage of the algorithm, each process initializes its local part of the input vector, that is defined by the index of the first element assigned to the local process named $myRange.begin$, and the index of the last one named $myRange.end$. The type *Range* represents a contiguous subdomain of indexes expressed as a pair of natural numbers $\langle begin, end\rangle$. Our example updates only the $Size/10$ neighbors of the main element in the input array (see the *Nelemt* variable initialization on line 19). In the second stage, the program calculates the range of elements to update (*first*, *last*). The third stage redistributes the selected range of elements to be updated in a balanced output array, evenly distributed, among the available processes (Step 3 in the left of Fig. 10.1). Finally in the stage 4, each process updates its new local part.

The data redistribution (stage 3) could be skipped, but it is desirable to balance the computational load. For example, if we execute the application with $Size = 10\,000$, and $P = 4$ without redistributing the data, as long as we apply a function only on a tenth of the elements (1 000), the computation will be performed only by two processes (at most), and we will not exploit all the nodes in the computation. To achieve a better load balance we can perform a data redistribution. In the case of $Size = 10\,000$, and $P = 4$, the elements to be computed (1000) are redistributed and each process will compute 250 elements in a balanced way.

On the right of Fig 10.1, we show the motivating algorithm using one of the operators proposed in this work (that performs a transparent data redistribution operation), in terms of a *Mapping function*, $L(id, Size, P)$, that returns the range of indexes to be mapped to the process id. As we observe in the figure, the programming effort is highly reduced avoiding the need of dealing with all the necessary communications details to balance the computational load.

Data redistributions are appropriated when the computation is performed in a unbalanced way according to the initial data distribution. They are convenient when the overhead produced by the communications needed to perform the data redistribution is expected to be less than the potential performance gain obtained by a better load balance. On the other hand, there are algorithms based on recursive, divide & conquer, or similar paradigms, for example *QuickSort*, which always implies dynamic modifications or subselections of array structures. Thus, data redistributions are totally necessary for executing these algorithms on distributed-memory systems.

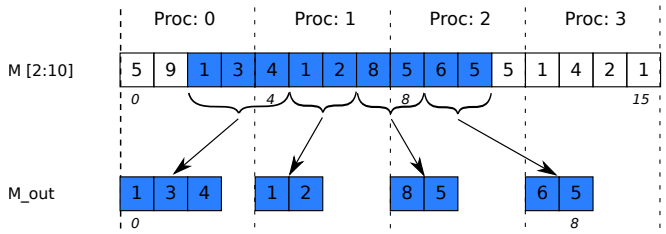

Figure 10.2: Data redistribution performed by the *ArrayRemapRange* operator. In this case the call to the operator is $M\_out = ArrayRemapRange(M, \langle 2, 10\rangle, L)$.

10.3 Proposal: Redistribution operators

In this section we describe four new high-level operators to perform array-data redistributions at runtime. They can be freely combined, even recursively, to transparently implement the communication structures of a wide range of array applications. Our operators receive an already-distributed array and different compulsory or optional parameters to select subdomains of the original array in different ways. The selected elements are redistributed across the whole range of available processes using a mapping function that assigns indexes to processes according to a given policy. Along the chapter, we name A_L to the set of arrays distributed using the mapping function L. For simplicity, the L mapping function used in all the figures of the chapter is a function for homogeneous load balance, that assigns contiguous blocks of indexes to each process. The current library implementation contains several mapping functions that can be chosen using their names as parameters in the corresponding functions.

10.3.1 *ArrayRemapRange*: Remap of an array range

The possibility of redistributing only a given selection or indexes range of the original array is appropriated for the improvement of the load-balance (such as the case of our motivating example). This first operator selects a range of an already distributed input array, and copy the selected elements in a new distributed output array. The interface of this operator is the following:

$$ArrayRemapRange : A_L\langle type\rangle, Range, L' \rightarrow A_{L'}\langle type\rangle$$

Both arrays (input and output) are distributed among the different processes not necessarily using the same mapping function. Several data communications per process can be needed to perform the data movement. Figure 10.2 shows a visual representation of how this operator works when the range $\langle 2, 10\rangle$, and a contiguous-blocks mapping policy are selected.

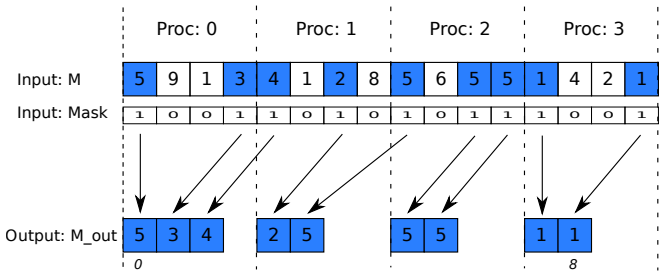

Figure 10.3: Data redistribution performed by the *ArrayRemapMask* operator. In this case the call to the operator is $M\_out = ArrayRemapMask(M, Mask, L)$.

10.3.2 *ArrayRemapMask*: Remap of an irregular selection using a mask

There are cases where the data that we want to select are not contiguous in memory. We propose a method based on *masks* to select this kind of sparse subdomains. The goal of this operator is similar to the *ArrayRemapRange* operator but using a mask to select the desired elements to be remapped. The mask is a boolean array with the same size than the input array. The input mask has to be also distributed using the same partition policy than the input array. This ensures that the mask value, for any given index, is mapped in the same process that its corresponding data element of the array. The output array can use the same or a different mapping function. The interface of this operator is the following:

$$ArrayRemapMask : A_L\langle type\rangle, A_L\langle bool\rangle, L' \rightarrow A_{L'}\langle type\rangle$$

The operator will select the elements whose associated value on the mask is 1 (true), the other ones are discarded. Figure 10.3 shows a visual representation of how this operator works.

10.3.3 *ArrayDivide*: Dividing an array in several balanced parts using a multivalued mask

This operator is designed to tackle recursive, divide & conquer, and similar applications that need to split the data in several groups, redistributing each group across processes independently. The operator receives an integer mask. The elements with the same natural value in the mask will be stored and redistributed in an independent output array. Thus, the output is a collection of arrays. The processes assigned to each output array are determined by a new kind of function that assigns a subset of the processes indexes $[0..P-1]$ to a different group, $T : \mathbb{N} \rightarrow P' \subset \{0, ..., P-1\}$. T is received as parameter by the operator. L' is used to redistribute each group across its assigned subset of processes. This operator is an

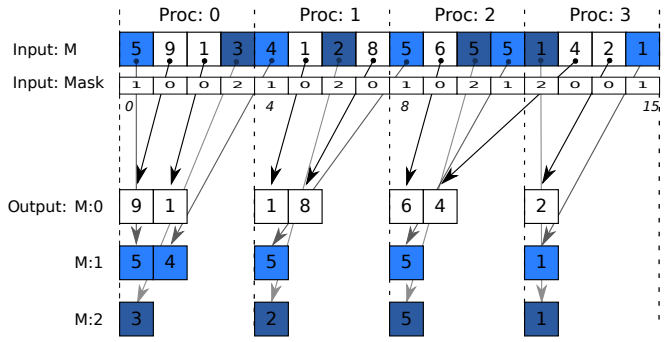

Figure 10.4: Data redistribution performed by the *ArrayDivide* operator. In this case the call to the operator is $M^* = ArrayDivide(M, Mask, L, T)$. The chosen T function for this example assigns to every group all the processes.

extension of ArrayRemapMask with several output arrays (equivalent to several simultaneous instances of ArrayRemapMask), and adding the T parameter that introduces the possibility of mapping each group to a different subset of processes. The definition of this operator is the following:

$$ArrayDivide : A_L\langle type\rangle, A_L\langle int\rangle, L', T \rightarrow \langle A^0_{L'}\langle type\rangle, ..., A^n_{L'}\langle type\rangle\rangle$$

In Fig. 10.4 we show an example of how this operator works for a specific mask with three group numbers (0, 1, 2), and a T function that assigns all the processes to every group. Thus, every output array is distributed in a balanced way across all processes.

10.3.4 *ArrayMerge*: Merging array parts

This fourth operator is designed to merge several distributed arrays in one. This operator receives a collection of arrays and concatenates them in one distributed output array. The output array is distributed in terms of the results of a mapping function and a processes assignment function T. The interface of this operator is the following:

$$ArrayMerge : \langle A^0_L\langle type\rangle, ..., A^n_L\langle type\rangle\rangle, L', T \rightarrow A_{L'}\langle type\rangle$$

Figure 10.5 shows a visual representation of how this operator works.

As another example of the use of the *ArrayDivide* and *ArrayMerge* operators, the *QuickSort* algorithm can be programmed in parallel using these operators (see Fig. 10.6). The algorithm divides a large array into two smaller sub-arrays: the elements lower, and the elements higher, than a pivot element. The program creates a mask that keeps the information about which element is lower or higher than the pivot (0 if the value is lower, or 1 if it is higher, as we see in the *Divide* stage). Using this mask, the program calls the *ArrayDivide* operator, which

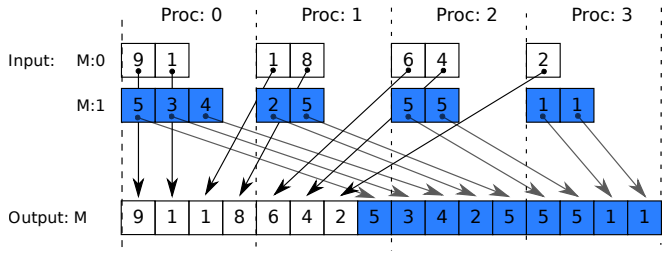

Figure 10.5: Operation performed by the *ArrayMerge* operator. In this case the call to the operator is $M = ArrayMerge(M^*, L, T)$. The chosen T function for this example assigns all the processes to the output array.

performs the data repartition. After that, the algorithm recursively calls again the *QuickSort()* function for each group array (see *Recursion* stage). For proper load balance, in *QuickSort* we use a T function that assigns to the group of each array a number of processes that is proportional to the array size. In the figure, for the first group array, the function T returns the processes $0, 1$, and the processes $2, 3$ for the second group array. After several recursive stages, the array has been splitted in several independent subarrays, distributed across the nodes, that have the sorted data. when exiting the recursion, the algorithm merges all the sorted subarrays using the *ArrayMerge* operator.

10.4 Implementation of the operators

In this section we present the extensions developed in Hitmap in order to support the new operators proposed.

10.4.1 Supporting data redistributions at Hitmap runtime level

We need to provide Hitmap with the necessary features, to develop the four operators. We introduce in Hitmap a new function named *localRange(Tile, Shape)*. It receives a distributed tile structure, and a selection range in global coordinates. It returns a *hit_shape* object representing the part of the input range that is allocated in the local process. For example, in Fig. 10.2, the function *localRange(M, [2:10])* returns for the process 0 the shape that selects its last two local elements, and for process 2 the shape that contains its first three local elements.

We extent HitLayout objects to be used for representing the T functions used in the *ArrayDivide* and *ArrayMerge* operators. We also develop in Hitmap a generic redistribution communication pattern constructor (*hit_patRedistribute()*). It receives two already distributed arrays (that in new versions of Hitmap contain a reference to their respective layout functions

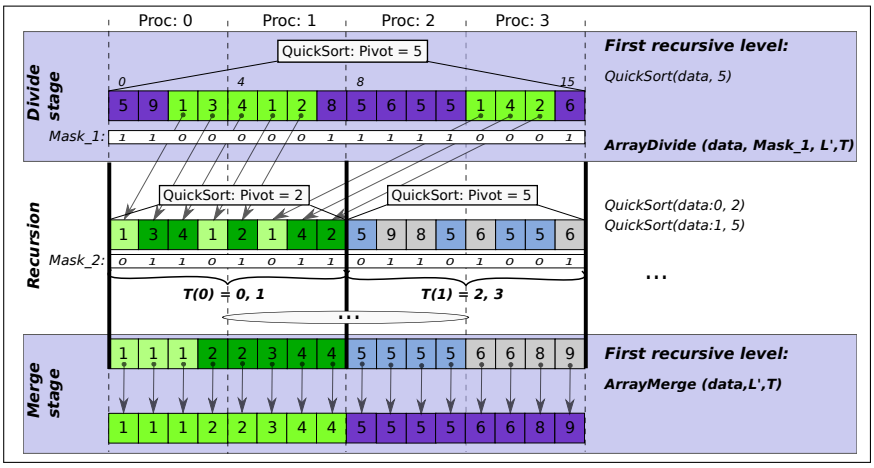

Figure 10.6: Sequence of operations performed in the QuickSort algorithm in a distributed-memory system using the *ArrayDivide* and *ArrayMerge* operators. T function assigns to each group array a number of processes that is proportional to the size of the group array.

L and L', originally used to distribute their domains). The constructor simply traverses the process-identifiers space with two loops. In the first loop, we compute the intersections of the result of applying L at the local process, with the result of applying L' at each remote process, to calculate the indexes of data to be sent. In the same way, the second loop computes the inverse intersections, applying L' at the local process, and L at each remote process to calculate the data to be received. The loops traverse the process identifiers in a cyclic way, starting at the local identifier plus 1; $(myRank + 1)\ mod\ P$. This generates a skewed communication scheme, that helps in reducing communication saturation bottlenecks on specific processes.

10.4.2 Implementation of the new operators

We implement the new operators as C macro-functions in the Hitmap library. For the rest of the section we represent ranges by *HitShape* objects, independently of the number of dimensions, and data structures by *HitTile* objects.

- **ArrayRemapRange**: The function prototype of this operator is:

```
ArrayRemapRange(tileIn, tileOut, rangeIn, rangeOut, baseType);
```

  The parameters represent: *tileIn* the data structure to be redistributed, *tileOut* the output data structure with the selected data, *rangeIn* the range of data to be selected

```
/* AUXILIAR MACROS */
#define NORMALIZE(TOK) NORMALIZE_ ## TOK
#define NORMALIZE_int HIT_INT
#define NORMALIZE_float HIT_FLOAT
#define NORMALIZE_double HIT_DOUBLE

/* ArrayRemapRange OPERATOR */
#define ArrayRemapRange(tileIn, tileOut,                              \
                        rangeIn, rangeOut, baseType){                  \
    /* STEP 1: DECLARE AUXILIAR VARIABLES */                           \
    HitTile_##baseType _TT_In, _TT_Out;                                \
    HitShape _TT_shape_Out,  _TT_shape_In;                             \
    /* STEP 2: EXTENT RANGES WITH NECESSARY HALOS */                   \
    int i;                                                             \
    for(i=0;i< hit_shapeSigDims(rangeIn);i++) {                        \
       int diff = hit_tileDimCard(tileOut,i) -                         \
                  hit_shapeSigCard(hit_layShape(tileOut.Layout),i);\
       hit_shapeDimExpand (range1, i, HIT_SHAPE_END, diff);            \
       hit_shapeDimExpand (range2, i, HIT_SHAPE_END, diff);            \
    }                                                                  \
    /* STEP 3: SELECT THE DATA CORRESPONDING TO EACH PROCESS */        \
    _TT_shape_In = localRange(tileIn, rangeIn);                        \
    _TT_shape_Out = localRange(tileOut, rangeOut);                     \
    hit_tileSelectArrayCoords (&_TT_In, &tileIn,_TT_shape_In);         \
    hit_tileSelectArrayCoords (&_TT_Out, &tileOut,_TT_shape_Out);      \
    /* STEP 4: PERFORM THE DATA REDISTRIBUTION */                      \
    HitPattern redis= hit_pattern (HIT_PAT_UNORDERED);                 \
    redis = hit_patRedistribute( (tileIn).Layout, (tileOut).Layout,\
                                 &_TT_In, &_TT_Out,                    \
                                 rangeIn, rangeOut,                    \
                                 NORMALIZE(baseType) );                \
}
```

Figure 10.7: Internal code of the *ArrayRemapRange* operator along with some auxiliary macro functions.

in *tileIn*, *rangeOut* the range where data will be in *tileOut* after the redistribution, and *baseType* is the name of the native or structured type.
This operator is used as baseline in the rest of operators development. Thus, we describe also its internal code. Figure 10.7 shows it. It first declares the necessary variables. In the cases that the input array has been modified by the programmer, including in the local domain overlapped parts with other remote domains (like halos in Stencil computations), the operator reproduces the same halos in the output array, by comparing the allocated domain with the domain assigned by the mapping function (step 2). The data moving to the halos is not included in the operator, as it depends on the program stage in which the operator is invoked.
After that, the program calculates and selects from the input array, the part of the selection range in the local process (step 3). The last step creates and executes the pattern containing the needed communications. The mapping functions used in the redistribution are those used to created the data structures. They are represented by *HitLayout* objects, and are kept as meta-data in the own *HitTile* data structure.

- **ArrayRemapMask**: The function prototype of this operator is:

```
ArrayRemapMask(tileIn, tileOut, maskIn, baseType);
```

  The parameters represent: *tileIn* the data structure to be redistributed, *tileOut* the output data structure with the selected data, *maskIn* the mask with the indexes of the data structure to be selected in *tileIn*, and *baseType* the base type of the data structure. Internally, this function code selects for each process the data elements whose mask value is 1 in the local process. In this case, to select the data, we generate a loop that traverses the local domain analysing the mask to identify the selected elements. It copies contiguously the selected data elements in an auxiliary array with contiguous memory. After that, a collective reduction communication is performed for sharing the information about the number of elements to copy for each process. Finally, the redistribution is performed using ranges as in the first operator.

- **ArrayDivide**: The function prototype of this operator is:

```
ArrayDivide(grouping, tileIn, tileOut, maskIn, baseType);
```

  The parameters represent: *grouping* a HitLayout object representing the T function (remind the operator definition), and containing the information of the number of natural values in the mask, *tileIn* the data structure to be redistributed, *tileOut* the output data structure with the selected data, *maskIn* the mask with the indexes of the data structure to be selected in *tileIn*, and *baseType* the base type of the data structure. The internal code creates a collection of arrays, where each array stores the elements that belong to the same array, using the same methodology that the *ArrayRemapMask* operator. However, this operator also stores in the meta-data of the data structures the global index domain of the original array. This last feature enables the use of the *ArrayMerge* operator.

- **ArrayMerge**: The function prototype of this operator is:

```
ArrayMerge(grouping, tilesIn, tileOut, baseType);
```

  The parameters represent: *grouping* the HitLayout object representing the T function, and containing the information of how concatenate the collection of input arrays, *tilesIn* the collection of arrays to be concatenated, *tileOut* the output array, and *baseType* the base type of the data structure. The internal code of this operator calls the Hitmap redistribution function for each input array, relocating the data in their corresponding ranges of the single output array. The ranges are calculated using the meta-data with the information about the index space on the original array, that was set by the *ArrayDivide* operator.

10.5 Experimental studies

We conduct several experimental studies to verify the efficiency of the resulting codes that use the proposed operators, in terms of runtime execution and development effort.

10.5.1 Experimental platform and setup

For the performance studies, we execute the experiments in CETA. It is a hybrid cluster that belongs to CIEMAT and the Spanish government. The cluster nodes are connected by Infiniband technology, and each one has two Intel Xeon 5520 CPUs at 2.27 GHz, with 4 cores each. Using 16 nodes of the cluster, we exploit up to 128 computational units. We have compiled the codes with the GCC v4.8.3 compiler, using the optimization flag *-O3*. We use *mpich3* v3.1.3 as MPI implementation. We execute all the experiments ten times, registering the average total execution times. We use a static mapping policy, associating one MPI process to each processing element. For all the routines and examples tested, we always use a mapping policy of contiguous balanced blocks, because of the homogeneous execution platform. For the development effort comparison, we use three classical development effort metrics: COCOMO lines of code, McCabe's cyclomatic complexity [86], and Halstead development effort [63].

10.5.2 Applying the operators: case studies

In order to validate our approach, we implement the following case studies using the proposed operators:

- **STL Benchmarks**: The STL Library is a well-known supporting tool for developers [122], that includes many useful algorithms. During the last years, many works have presented parallel versions of this library [50, 120, 123], as well as new parallel programming models that support the development or the use of this library in parallel [129].

Table 10.1: Summary of the implemented STL routines for one dimensional numeric arrays, for distributed-memory systems, using the new four operators.

Algorithm Class	Function Call(s)
Embarrassingly Parallel	all_of, any_of , none_of, copy, copy_if, copy_n, count, count_if, fill, fill_n, for_each, generate, generate_n, replace_if, transform, swap_ranges
Find	find, find_if, find_end, find_first_of, adjacent_find, mismatch, equal
Search	search, search_n, lower_bound, upper_bound
Numerical Algorithms	min_element, max_element, minmax_element, accumulate, adjacent_difference, inner_product, rotate, rotate_copy
Partition	is_partitioned, partition, partition_copy, partition_point
Merge	merge, inplace_merge
Sort	quickSort, is_sorted, is_sorted_until, partial_sort_copy,
Complex Set Operations	remove_if, remove_copy_if, set_difference, set_symmetric_difference, set_intersection, set_union, unique, unique_copy

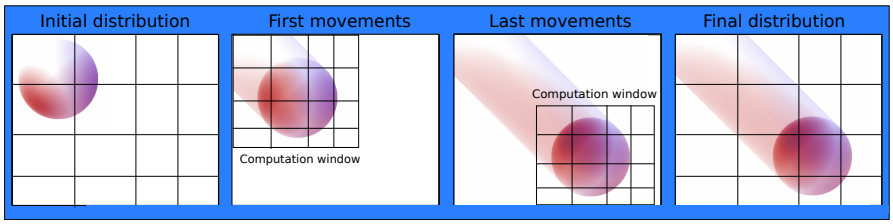

Figure 10.8: Consecutive applications of the RayTracing algorithm on a moving sphere.

We have implemented using the operators, the routines summarized in Tab. 10.1. However, in this work we only show the results of four specific routines. They cover the different kinds of data redistributions that together can support all the other implemented STL algorithms. The results are representative and can be extrapolated to the other STL routines tested. The routines we choose are:

- **for_each**: This example updates a range of elements of an array. The program performs N operations for each element on the selected range of the array, being N the array size. We use the *ArrayRemapRange* operator. This kind of data redistribution appears in most of the algorithms in the STL.

- **find**: This routine searches the first position in a range of an array, that fits with a specified condition. It applies a function on each element before a global reduction. This example also use a data redistribution that remap a selected range of a distributed array. It is another example of the use of the *ArrayRemapRange* operator but followed by a conditional and a reduction communication.

- **unique_copy**: This example copies the elements of a range of an array to a second array, skipping the consecutive duplicates. The size of the second data structure contains the elements copied from the first data structure. We use the *ArrayRemapMask* operator to only select data elements that have to be copied. Each process fills its assigned part of the mask previously to the operator invocation, comparing each element with its neighbor. In the experimental study we initialize with the same value a 20% of contiguous data elements. Thus, all these elements except the first one are eliminated from the output array.

- **quickSort**: This example sorts the elements of an array. To implement this algorithm in a distributed-memory system, it is necessary to perform a sequence of recursive data redistributions. We use the *ArrayDivide* and *ArrayMerge* operators to part and merge the pivoted arrays at each recursion level.

- **RayTracing Algorithm**: Raytracing is a technique for generating an image in a 3D scene calculating trajectories of light rays through pixels in a view plane [74, 75]. This technique produces a very high degree of photorealism. However, its high computational cost makes its use prohibitive in real-time applications for complex scenarios. RayTracing is an embarrassing parallel application, where a scene-dependent workload partition could achieve huge improvements on performance.
  We select an example program that applies the RayTracing algorithm on a sequence on scenes with an object in movement. Figure 10.8 shows the behavior of the study case. In the figure, the scene is a sphere, where the shadow of this sphere is enlarged on each step. In the first stage (*Initial distribution*), the image is distributed among the active processes (16 in the figure) using a 2D irregular partition policy using block sizes based on the computational power of the execution machines. For this first distribution, there is no knowledge about the scene, so the whole image is evenly distributed according to the computational parameters of the machines. As we observe in the example image, there are only four processes initially involved on computing the raytracing on the interest zone surrounding the sphere. The computation window is detected on each moment, and a data redistribution is performed in order to balance the work load, as we see in the First and Last movements of Fig. 10.8. In order to avoid a data redistribution on each movement, the computation window is bigger than the real scenario. Thus, the data redistribution only is performed when the actual pixels that need computation are close to the boundaries.
  For this application, we develop a code based on the sequential code of [114]. Our case study scene has a sphere with a size of one eighth of the image size, and we execute a number of movements equal to a tenth of the image size in a diagonal direction as in the image. As for data redistributions, in our implementation we use the *ArrayRemapRange* operator, selecting a 2D domain. Using the current scene and parameters, the program needs to perform 6 redistributions. In this experimental study, we use a mapping policy of contiguous balanced blocks, because of the homogeneous execution platform.

10.5.3 Impact of redistributing workload on performance

Although the positive effect of the data redistributions on several applications have been already studied [75, 140], in this section, we present a performance study on a distributed-memory system to show these positive benefits. We test the *for_each* routine on an array of 10^6 elements with different range selection cases. It is an example that easily allows the exploration of the effects on performance related to the variation of the amount of load redistributed, and its location on the original domain.

We test three kind of codes. The first kind of code is a Hitmap implementation that redistributes the data in the selected range across all the processes to balance the computational load using the operators (named *Proposal* in figures). The second is a Hitmap implementation that does not include data redistributions (named *Hitmap* in figures), and each process works

with its originally mapped data that are in the selected range (if any). Both implementations use the same sequential functions and semantic structure, so we only see the performance penalty or gain that comes from using our data-redistribution operators, that is the focus of our study. Moreover, we add a third kind of code. It is a manually developed and optimized MPI reference, also containing manual data redistributions (named *Ref MPI* in figures), to show the penalty performance produced by our abstractions and approach.

We design the experiments in order to study the impact of two parameters in the data-redistribution operations:

1. The amount of data selected from the original array where applying the routine. We perform the experimentation selecting 20%, 50% and 80% of the data in the whole vector.

2. The place in the original array where the range of data is selected. Data redistributions can obtain different performance in function of the number of processes actually implied in the communications. Thus, we perform the experimentation selecting the data in three ways: (1) Selecting a range of data chosen from the beginning of the array (*Left*), (2) selecting the data at the end of the array (*Right*), and (3) selecting the data, with the center of the selected range at the middle point of the whole array (*Center*).

Figure 10.9 shows the performance obtained for the different versions and parameters of the *for_each* routine in CETA. First, we see that the penalty performance of our approach using the operators compared with a MPI reference is negligible (both lines are overlapped in all the plots). Second, we observe the impact of the load balance obtained with the data redistributions. When the data selection is 80% of the whole array, redistributing the data has not a big performance impact. However, when the amount of data selection is low (20% or 50%), the performance obtained by the load-balanced codes is significantly better than in the codes which do not use it, despite the extra communication cost of the data reallocation forth and back. Our operators redistributes the data that need computation, avoiding idle processors and creating load balance.

10.5.4 Using the STL library for analyzing the four operators

The previous section showed the importance of the data redistributions. The goal of the next is to analyze the behavior of the different kinds of operators presented in this work. We analyze and compare our proposal with MPI, in terms of execution time and development effort for the chosen STL routines, that cover the different kinds of data redistributions.

In Tab. 10.2 we show several development effort measures, comparing the described STL routines, coded in Hitmap with the new operators (*Proposal*), or coded directly in MPI (*Ref MPI*). For the QuickSort we use as baseline the implementation presented in [117, 138]. We see that using the operators, the measures for the chosen metrics are highly reduced. This indicates a clear simplification of the programmability for the developer.

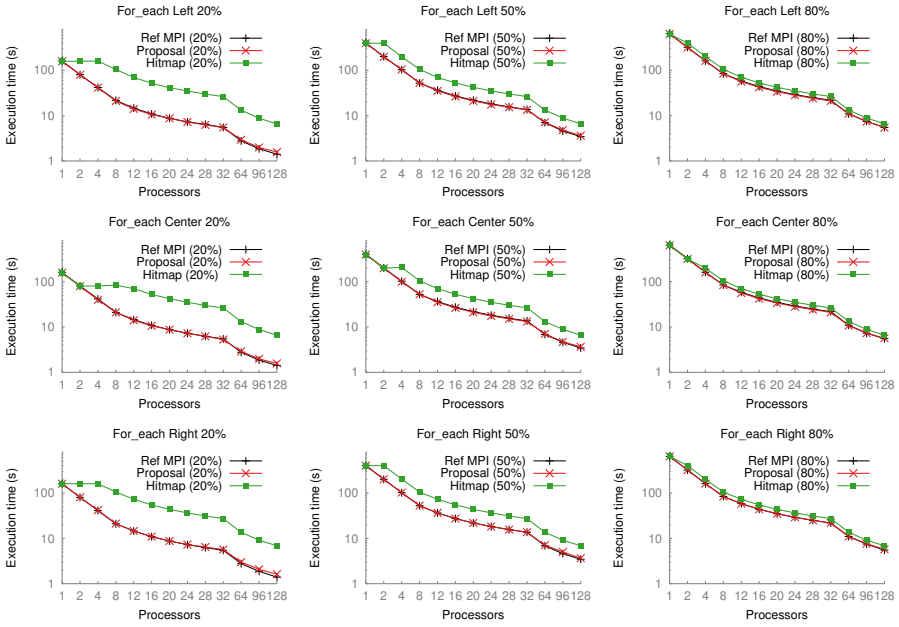

Figure 10.9: Performance scalability results (in seconds) for the *for_each* algorithm in CETA, the distributed-memory system (logarithmic scale). $Size = 1000000$.

Table 10.2: Measures of development effort for the STL study cases, comparing our proposal with MPI.

Benchmark	Unique copy		QuickSort	
Measure	**Ref MPI**	**Proposal**	**Ref MPI**	**Proposal**
N. Lines	266	117	237	156
Mccabe C. C.	42	4	47	25
Halstead D. E.	1,763,360	306,887	1,411,782	575,337

Benchmark	For each		Find	
Measure	**Ref MPI**	**Proposal**	**Ref MPI**	**Proposal**
N. Lines	135	72	151	68
Mccabe C. C.	19	3	23	3
Halstead D. E.	517,153	110,692	786,800	105,709

On the other hand, table 10.3 shows the execution times obtained for the STL study cases (the *for_each* routine was already studied in the previous section). We show the results when the routines are executed with 128 MPI processes in CETA. We observe that the performance obtained by our approach is similar to the reference version. These examples have a very low computational load. Thus, the main advantage of redistributing the data structures comes from keeping them distributed among several nodes, instead of reducing the computation time.

Table 10.3: Measures (in miliseconds) of performance for the STL study cases comparing our proposal with MPI, using 128 MPI processes in CETA (the *for_each* routine results were presented in Fig. 10.9).

Routine	Size	Ref MPI	Proposal
Find	10^5	66.253	66.983
Unique copy	10^5	51.015	51.705
QuickSort	10^5	1.023	1.098

Table 10.4: Measures of development effort for the RayTracing algorithm, comparing our proposal with MPI.

Routine	MPI	Hitmap	Proposal
N. Lines	322	255	302
Mccabe C. C.	32	14	15
Halstead D. E.	3,150,194	1,912,643	3,224,414

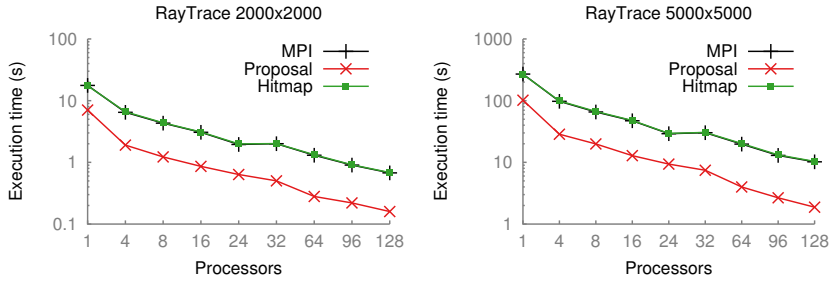

Figure 10.10: Performance scalability results (in seconds) for the *RayTracing* algorithm in CETA with different image sizes (logarithmic scale).

10.5.5 Evaluating the use of the proposal on a real-world application: Raytracing algorithm

In this study we analyze the use of the proposed operators in a 2D real-world application, the Raytracing algorithm.

We compare three codes. The first one is a pure MPI reference code, with no data redistributions (*MPI*). The second one is a Hitmap code, also with no data redistributions (*Hitmap*). The third code is a Hitmap code, where data redistributions are applied using the approach presented in this work (*Proposal*).

Table 10.4 shows the development effort measures of the studied codes. We observe that using the operators the development effort needed to program a code is slightly increased compared with a pure Hitmap code (with no data redistributions), but lower that the MPI reference code (with no data redistributions).

However, the performance is highly improved, as we see in Fig. 10.10, where the code with the operators achieves the best results. In this figure, we also observe that the codes with no data redistributions, Hitmap code and the MPI reference, obtain the same performance (lines overlapped in the figure).

Our operators abstract to the programmer all the data-redistribution implementation issues, related to the data partition, data ownership, and data communication. Moreover, despite the potential overhead derived from dealing with distributed data in an abstract way, our approach produces a good scalability, and a negligible penalty performance compared with the MPI implementations.

10.6 Summary

This chapter presents four array data-redistribution operators to efficiently implement distributed-memory algorithms, making the data partition, relocation and data movement transparent to the programmer. Our proposed operators provide programming abstractions to manage data redistributions. We also present the application of the operators in a real-world application (RayTracing), and in many algorithms of the C++ STL library.

With our proposal, the programmer does not need to deal with data-redistribution implementation issues that are not related with the algorithms but are key in terms of performance. Experimental results show that our proposal achieves the same performance than optimized MPI codes, meanwhile the programming effort is highly reduced. Future work includes the exploitation of the proposed operators in automatic code generators.

Chapter 11

Conclusions

This PhD. Thesis addresses several main problems related to the parallel programming for highly heterogeneous and distributed systems. The contributions of the first part of the Thesis enable the creation of abstract programming structures for coordination code. They allow the programming of codes portable across different devices, accelerators, and architectures. At the same time they maintain maximum efficiency thanks to the internal use of lower-level or specific-vendor features and tools. In the second part, we dealt with many issues concerning the distributed-memory systems. Our proposals, based on moving compile techniques to runtime, show the way to support a wider range of applications in the future compilers and frameworks.

The combination of these solutions will lead to a new approach for heterogeneous systems with several distributed nodes . It will be achieved by the integration of the techniques presented in the second part, into the programming model proposed in the first part.

11.1 Summary of contributions

This section summarizes the contributions of this PhD. Thesis and the published articles derived from this research work. We present the summary of the contributions divided according to the two parts of the Thesis.

11.1.1 Simplifying the programming on accelerator-based heterogeneous systems

Programming solutions for an efficient deployment on heterogeneous systems is a very complex task that relies on the manual management of the different computing platforms, also considering architectural details. During the first part of the Thesis, we have presented some proposals in order to achieve the goals proposed for it in the introduction chapter. In this section we summarize the contributions introduced to achieve each goal, including the publications where each one has been presented to the community.

1. We have presented an abstract entity (Controller), introduced as a library of functions, that allows the programming of portable coordination codes for groups of CPU cores and GPUs. It also allows the programming of simple generic kernels that can be executed in both devices, and does a transparent management of data movements across different memory hierarchies.
   This accomplishes **Goal 1**: Design and develop a programming model to unify the parallel programming of CPUs and GPUs.
   Publication:

   > **Journal JCR Q2:** [99] Ana Moreton–Fernandez, Hector Ortega–Arranz and Arturo Gonzalez–Escribano. 'Controllers: An abstraction to ease the use of hardware accelerators'. In: *The International Journal of High Performance Computing Applications*, 2017. 2017. DOI: 10.1177/1094342017702962. eprint: http://dx.doi.org/10.1177/1094342017702962. URL: http://dx.doi.org/10.1177/1094342017702962

2. We have introduced in the Controller model the support for a new type of device, the Intel Xeon Phi (MIC) accelerators. We have done it without modifying the semantics or interface of the first Controller proposal. We mixed the internal execution model used for CPU cores, with the internal communications model used for GPU devices of the original Controller, translating the operations to the specific idioms to efficiently use the Xeon Phi device.
   This fulfils **Goal 2**: Support for new kind of accelerators such as the Xeon Phi Coprocessor.
   Publication:

Conference CORE A: [100] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano and Diego R Llanos. 'Supporting the Xeon Phi Coprocessor in a Heterogeneous Programming Model'. In: *European Conference on Parallel Processing*. Springer, Cham. 2017, pp. 457–469

3. We have introduced an abstract entity (the MultiController) that coordinates in a transparent way the same operations proposed in the Controller model, over several Controller entities. It is independent of the kind of device to which each internal Controller entity is associated.
   This solves the problem proposed in **Goal 3**: Design and develop an adaptable, simple, and efficient programming model for multiple heterogeneous devices.
   Publication:

   Journal JCR Q3: [96] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Multi-device Controllers: A library to Simplify Parallel Heterogeneous Programming'. In: *International Journal of Parallel Programming*, 2017, pp. 1–20. Springer, 2017

11.1.2 Automatizing the data management for distributed-memory spaces in heterogeneous systems

The parallel programming of applications with data dependences has been a huge challenge for programmers, specially in execution systems with separated memory spaces, where this kind of applications imply data communications. In the second part of the Thesis, we presented a set of techniques that abstract many data-communication implementation issues to the programmer. They improve the approaches of the literature being independent of compile-time decisions, generating exact coarse-grained communications, or producing balance workload. The contributions presented in this part can be summarized as follows:

4. A study of the run-time cost of the codes generated for data communication in distributed-systems by a state-of-the-art tool based in the Polyhedral model. This study analyses the impact of code transformations introduced at compile time, and its limitations. We also propose a simplification that removes one of them, eliminating from the complexity bound of the execution time one factor that grows linearly with the input data size.
   Publication:

   Conference CORE B: [97] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'On the run-time cost of distributed-memory communications generated using the polyhedral model'. In: *High Performance Computing & Simulation (HPCS), 2015 International Conference on*. IEEE. 2015, pp. 151–159

5. As a basic tool to develop and test new transformation techniques, we proposed a programming framework to simplify the expression of parallel codes of arbitrary computation grain, in terms of SPMD blocks. This proposal is designed to simplify the identification and extraction of data dependences, and the introduction of transformation techniques.
   Publications:

   International Conference: [8] Arturo Gonzalez-Escribano Ana Moreton-Fernandez and Diego R. Llanos. 'Trasgo 2.0: Code generation for parallel distributed- and shared-memory hierarchical systems'. In: *Compilers for Parallel Computing (CPC)*. London, 2015

   International Conference: [93] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'A New High-Level Parallel Portable Language for Hierarchical Systems in Trasgo'. In: *Computational and Mathematical Methods in Science and Engineering (CMMSE)*. 2015

6. We have presented a new technique that, using the data dependence information, is able to calculate at run-time the communications that should be performed across SPMD blocks to ensure the semantics of the parallel expressions. The technique works for partitions with arbitrary grain, and potentially irregular sizes on each part.
   This accomplishes **Goal 4**: Design and develop a runtime technique to automatically calculate aggregated communications for applications with affine expressions.
   Publications:

   Poster International School: [7] Arturo Gonzalez-Escribano Ana Moreton-Fernandez and Diego R. Llanos. 'Simple and Efficiente parallel programming for distributed-memory systems'. In: *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Fiuggy, Italy, 2016

   Journal JCR Q2: [94] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'A Technique to Automatically Determine Ad-hoc Communication Patterns at Runtime'. In: *Parallel Computing*, 2017. North-Holland, 2017

7. We have also presented a complete new technique, based in the same concepts as the previous one, that can take into account periodic conditions introduced in the data spaces in which the parallel application is operating. This shows the applicability of the same ideas to whole new type of application.
   This targets the **Goal 5**: Extend the technique to automatically calculate aggregated communications in applications with periodic domains.
   Publications:

Poster Conference CORE A: [90] Ana Moreton, Arturo Gonzalez-Escribano and Diego R. Llanos. 'A Runtime Analysis for Communication Calculation'. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. poster, 2017

Conference CORE C: [91] Ana Moreton-Fernandez and Arturo Gonzalez-Escribano. 'Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions'. In: *10th International Symposium on High-Level Parallel Programming and Applications (HLPP)*. Valladolid, Spain, 2017

Journal JCR Q3: [92] Ana Moreton-Fernandez and Arturo Gonzalez-Escribano. 'Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions'. In: *Concurrency and Computation: Practice and Experience*, 2018. Wiley, 2018

8. Finally, we have tackled an important problem when facing applications that dynamically change the computational load associated with the different parts of the distributed data domains. We have proposed four abstract data-redistribution operators that can be used to rebalance the workload in multiple types of applications. To show its applicability, we have programmed with them most of the routines of the C++ STL library, and simple variable load programs, including a RayTracing application which focuses the computational workload in a object which moves across a scene.
This fulfils the **Goal 6**: Propose an abstraction to simplify data redistributions.
Publications:

   Conference CORE C: [6] Arturo Gonzalez-Escribano Ana Moreton-Fernandez and Diego R. Llanos. 'Four abstract array distribution operators'. In: *9th International Symposium on High-Level Parallel Programming and Applications (HLPP)*. Muster, Germany, 2016

   Journal JCR Q2: [98] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Operators for Data Redistribution on the STL library and RayTracing Algorithm (Submitted)'. In: *Journal of Parallel and Distributed Computing*, 2018. Elsevier, 2018

11.2 Answer to the research question

This section answers the research question presented in Chapter 1:

> *It is possible to introduce: (1) Simple abstractions to make transparent and uniform the program deployment and coordination on parallel computational units of different types and architectures, and (2) new runtime techniques that take into account the*

data dependences in order to automatically coordinate the data communication across disjoint or distributed memory spaces, in a transparent way for the programmer, once the features of the execution machine are known?

During this dissertation we have presented different proposals with the final goal of answering in an affirmative way this research question. We presented an adaptable and simple programming model for heterogeneous systems composed by a node with several accelerators, and we have proposed several automatic techniques to make data transfers transparent in heterogeneous systems with several distributed nodes or computational units with a separate space memory.

11.3 Future Directions

The approaches delivered in this dissertation lead clearly to a new proposal. Using as foundation the programming model presented at the end of the first part, and integrating into it the techniques presented in the second part, we propose as future work the design and development of a simple and adaptable programming model for highly heterogeneous distributed-memory systems. These execution platforms are composed by a distributed-memory system, where each host has associated several computational units (GPUs, XeonPhi, or groups of CPU-cores). The possibility of integrating all the developed methods and tools proposed in this Thesis in a simple framework to program parallel heterogeneous distributed-memory systems presents a new challenge that will simplify highly the development effort of the HPC programmers. In this new research project the mapping techniques should be used to automatically create distributed arrays across different computational units and coprocessors in a distributed-memory system, taking into account the different computation capabilities. The communication calculation techniques should be used to automatically determine the data movements across devices, and new techniques should be introduced in the Multi-Controllers model to integrate these data movement operations across remote multi-controller entities in a transparent way.

More future directions can be considered:

- *Study of the impact of changing the mapping policies on the overall performance.* The Multi-Controller library provides a simple method to change the data partition policy applied for distributing the computation. It is interesting to perform an study and to define an analytical runtime model, based on the computational features of the execution machine and of the exploited application, that can be used to determine the best partition policy and run-time configuration, in terms of performance.

- *New techniques for overlapping computation with communication.* Overlapping computation and communication impact nicely in the performance in many applications. However, it is difficult to determine and to manage the computation parts where

it is possible to do it. A relevant work for the community can be the development of automatic techniques able to define at runtime the computation part where it is possible to overlap communications and computation, and to effectively introduce the overlapping.

Bibliography

[1] W. Richards Adrion. 'Research Methodology in Software Engineering'. In: *ACM SIGSOFT Software Engineering Notes. Summary of the Dagstuhl Workshop on Future Directions in Software Engineering*, vol. 18, no. 1, 1993, pp. 36–37. ACM, 1993. DOI: 10.1145/157397.157399.

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick and Massimo Torquati. 'Targeting distributed systems in fastflow'. In: *European Conference on Parallel Processing*. Springer. 2012, pp. 47–56.

[3] Francisco Almeida, Domingo Giménez, José Miguel Mantas and Antonio M Vidal. *Introducción a la programación paralela*. Thompson Paraninfo, 2008.

[4] Saman P Amarasinghe and Monica S Lam. 'Communication optimization and code generation for distributed memory machines'. In: *ACM SIGPLAN Notices*. Vol. 28. 6. ACM. 1993, pp. 126–138.

[5] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato and Lawrence Rauchwerger. 'STAPL: An adaptive, generic parallel C++ library'. In: *Languages and Compilers for Parallel Computing*. Springer, 2001, pp. 193–208.

[6] Arturo Gonzalez-Escribano Ana Moreton-Fernandez and Diego R. Llanos. 'Four abstract array distribution operators'. In: *9th International Symposium on High-Level Parallel Programming and Applications (HLPP)*. Muster, Germany, 2016.

[7] Arturo Gonzalez-Escribano Ana Moreton-Fernandez and Diego R. Llanos. 'Simple and Efficiente parallel programming for distributed-memory systems'. In: *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Fiuggy, Italy, 2016.

[8] Arturo Gonzalez-Escribano Ana Moreton-Fernandez and Diego R. Llanos. 'Trasgo 2.0: Code generation for parallel distributed- and shared-memory hierarchical systems'. In: *Compilers for Parallel Computing (CPC)*. London, 2015.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams et al. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. 2006.

[10] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan and Guansong Zhang. 'The design of OpenMP tasks'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, 2009, pp. 404–418. IEEE, 2009.

[11] David Bailey, Tim Harris, William Saphir, Rob van der Winjgaart, Alex Woo and Maurice Yarrow. *The NAS Parallel Benchmarks 2.0*. Report RNR-95-020. 1995.

[12] Pavan Balaji. *Programming models for parallel computing*. MIT Press, 2015.

[13] Youcef Barigou and Edgar Gabriel. 'Maximizing Communication--Computation Overlap Through Automatic Parallelization and Run-time Tuning of Non-blocking Collective Operations'. In: *International Journal of Parallel Programming*, 2016, pp. 1–27. Springer, 2016.

[14] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan et al. 'The Habanero multicore software research project'. In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM. 2009, pp. 735–736.

[15] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev and Ponnuswamy Sadayappan. 'Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories'. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM. 2008, pp. 1–10.

[16] Cedric Bastoul. 'Code generation in the polyhedral model is easier than you think'. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. 2004, pp. 7–16.

[17] Mehmet E Belviranli, Laxmi N Bhuyan and Rajiv Gupta. 'A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures'. In: *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, 2013, p. 57. ACM, 2013.

[18] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Maria J. Garzaran, David Padua and Christoph von Praun. 'Programming for parallelism and locality with hierarchically tiled arrays'. In: *Proc. of the ACM SIGPLAN PPoPP*. New York, New York, USA: ACM, 2006, pp. 48–57.

[19] Barry W Boehm et al. *Software engineering economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[20] Dan Bonachea. 'GASNet Specification, v1. 1'. In: 2002. University of California at Berkeley, 2002.

[21] Uday Bondhugula. 'Compiling affine loop nests for distributed-memory parallel architectures'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM. 2013, p. 33.

[22] Uday Bondhugula. *PLUTO-an automatic parallelizer and locality optimizer for multicores*. on `http://pluto-compiler.sourceforge.net`. 2009.

[23] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron and Nicolas Vasilache. 'Tiling and optimizing time-iterated computations on periodic domains'. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 39–50.

[24] Uday Bondhugula, Vinayaka Bandishti and Irshad Pananilath. 'Diamond tiling: Tiling techniques to maximize parallelism for stencil computations'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, 2017, pp. 1285–1298. IEEE, 2017.

[25] Uday Bondhugula, Albert Hartono, J. Ramanujam and P. Sadayappan. 'A Practical Automatic Polyhedral Program Optimization System'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2008.

[26] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik and Olaf O Storaasli. 'State-of-the-art in heterogeneous computing'. In: *Scientific Programming*, vol. 18, no. 1, 2010, pp. 1–33. Hindawi Publishing Corporation, 2010.

[27] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston and Pat Hanrahan. 'Brook for GPUs: stream computing on graphics hardware'. In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 777–786.

[28] Javier Bueno, Xavier Martorell, Rosa M Badia, Eduard Ayguadé and Jesús Labarta. 'Implementing ompss support for regions of data in architectures with multiple address spaces'. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 359–368.

[29] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[30] Lynn E Cannon. *A CELLULAR COMPUTER TO IMPLEMENT THE KALMAN FILTER ALGORITHM*. Tech. rep. 1969.

[31] B.L. Chamberlain, S.J. Deitz, D. Iten and S-E. Choi. 'User-Defined Distributions and Layouts in Chapel: Philosophy and Framework'. In: *2nd USENIX Workshop on Hot Topics in Parallelism*. June 2010.

[32] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald and Ramesh Menon. *Parallel programming in OpenMP*. 1st ed. Morgan Kaufmann, 2001. ISBN: 1-55860-671-8.

[33] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[34] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun and Vivek Sarkar. 'X10: an object-oriented approach to non-uniform cluster computing'. In: *Acm Sigplan Notices*, vol. 40, no. 10, 2005, pp. 519–538. ACM, 2005.

[35] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar and Yonghong Yan. 'Integrating asynchronous task parallelism with MPI'. In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 712–725.

[36] Daniel Chavarría-Miranda and John Mellor-Crummey. 'Effective communication coalescing for data-parallel applications'. In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2005, pp. 14–25.

[37] Q. K. Chen and J. K. Zhang. 'A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA'. In: *ICISE'2009*. Dec. 2009, pp. 86–89. DOI: 10.1109/ICISE.2009.171.

[38] Michael Claßen and Martin Griebl. 'Automatic code generation for distributed memory architectures in the polytope model'. In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE. 2006, 7–pp.

[39] OpenACC Consortium. *OpenACC: Directives for Accelerators*. eng. WWW. on http://www.openacc-standard.org/. 2011--2015.

[40] Leonardo Dagum and Ramesh Menon. 'OpenMP: an industry standard API for shared-memory programming'. In: *IEEE computational science and engineering*, vol. 5, no. 1, 1998, pp. 46–55. IEEE, 1998.

[41] Usman Dastgeer, Johan Enmyren and Christoph W. Kessler. 'Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems'. In: *Proc. IWMSE'11*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 25–32. ISBN: 978-1-4503-0577-8.

[42] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar and Uday Bondhugula. 'Generating efficient data movement code for heterogeneous architectures with distributed-memory'. In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE. 2013, pp. 375–386.

[43] HV Deepika, NN Mangala and Sarat Chandra Babu. 'Automatic program generation for heterogeneous architectures'. In: *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*. IEEE. 2016, pp. 102–109.

[44] Javier Diaz, Camelia Munoz-Caro and Alfonso Nino. 'A survey of parallel programming models and tools in the multi and many-core era'. In: *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, 2012, pp. 1369–1386. IEEE, 2012.

[45] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek and Stanimire Tomov. 'HPC programming on Intel many-integrated-core hardware with Magma port to Xeon Phi'. In: *Scientific Programming*, vol. 2015, no. 9, 2015, pp. 1–11. Hindawi Publishing Corp., 2015.

[46] Heinz-Dieter Ebbinghaus, Jörg Flum and Wolfgang Thomas. *Mathematical logic, 2nd edn. Undergraduate Texts in Mathematics*. 1994.

[47] Wu-chun Feng and Pavan Balaji. 'Tools and environments for multicore and many-core architectures'. In: *Computer*, vol. 42, no. 12, 2009, pp. 26–27. IEEE, 2009.

[48] Javier Fresno Bausela. 'Supporting general data structures and execution models in runtime enviroments'. PhD thesis. Universidad de Valladolid, 2015. URL: `https://www.infor.uva.es/~jfresno/papers/jfresno_thesis.pdf`.

[49] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Blending Extensibility and Performance in Dense and Sparse Parallel Data Management'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, 2014, pp. 2509–2519. IEEE, 2014. DOI: `10.1109/TPDS.2013.248`.

[50] Leonor Frias and Johannes Singler. 'Parallelization of bulk operations for STL dictionaries'. In: *Euro-Par 2007 Workshops: Parallel Processing*. Springer. 2007, pp. 49–58.

[51] Tobias Fuchs and Karl Fürlinger. 'A Multi-Dimensional Distributed Array Abstraction for PGAS'. In: *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*. IEEE. 2016, pp. 1061–1068.

[52] Karl Fürlinger, Colin Glass, Jose Gracia, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb and Huan Zhou. 'DASH: Data structures and algorithms with support for hierarchical locality'. In: *European Conference on Parallel Processing*. Springer. 2014, pp. 542–552.

[53] T. El-Ghazawi, W. Carlson, T. Sterling and K. Yelick. *UPC : distributed shared-memory programming*. eng. Wiley-Interscience, 2003. ISBN: 978-0471220480.

[54] Arturo González-Escribano and Diego R Llanos. 'Trasgo: a nested-parallel programming system'. In: *The Journal of Supercomputing*, vol. 58, no. 2, 2011, pp. 226–234. Springer, 2011.

[55] Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno and Diego R. Llanos. 'An Extensible System for Multilevel Automatic Data Partition and Mapping'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, May 2014, pp. 1145–1154. May 2014. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.83.

[56] Ananth Grama, Anshul Gupta, George Karypis and Vipin Kumar. *Introduction to Parallel Computing*. 2º. Addison Wesley, 2003, p. 856. ISBN: 0201648652.

[57] Ivan Grasso, Simone Pellegrini, Biagio Cosenza and Thomas Fahringer. 'A uniform approach for programming distributed heterogeneous computing systems'. In: *Journal of parallel and distributed computing*, vol. 74, no. 12, 2014, pp. 3228–3239. Elsevier, 2014.

[58] William Gropp, Ken Kennedy, Linda Torczon, Andy White, Jack Dongarra, Ian Foster and Geoffrey C Fox. *The Sourcebook of Parallel Computing*. 2002.

[59] William Gropp, Ewing Lusk and Anthony Skjellum. *Using MPI : Portable Parallel Programming With the Message-passing Interface*. 2nd ed. MIT Press, 1999, p. 371. ISBN: 0262571323.

[60] William Gropp, Ewing Lusk and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 2014. Chap. 4.

[61] Trasgo Research Group. *Trasgo webpage*. URL: http://trasgo.infor.uva.es (visited on 19/05/2015).

[62] Michael Haidl and Sergei Gorlatch. 'PACXX: Towards a Unified Programming Model for Programming Accelerators using C++14'. In: *Proc. LLVM-HPC'14*. IEEE, 2014.

[63] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[64] Khaled Hamidouche, Joel Falcou and Daniel Etiemble. 'A Framework for an Automatic Hybrid MPI+OpenMP Code Generation'. In: *Proc. HPC'11*. Boston, Massachusetts: Society for Computer Simulation International, 2011, pp. 48–55.

[65] Tianyi David Han and Tarek S. Abdelrahman. 'hiCUDA: a high-level directive-based language for GPU programming.' In: *GPGPU*. Ed. by David R. Kaeli and Miriam Leeser. Vol. 383. ACM, 24th Mar. 2009, pp. 52–61. ISBN: 978-1-60558-517-8.

[66] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, Jagannathan Ramanujam and Ponnuswamy Sadayappan. 'Parametric multi-level tiling of imperfectly nested loops'. In: *Proceedings of the 23rd international conference on Supercomputing*. ACM. 2009, pp. 147–157.

[67] Michael T Heath. 'Scientific computing'. In: McGraw-Hill, 2001. Chap. 11.

[68] Jacob Hemstad, Ulf R Hanebutte, Ben Harshbarger and Bradford L Chamberlain. 'A Study of the Bucket-Exchange Pattern in the PGAS Model Using the ISx Integer Sort Mini-Application'. In:

[69] Pieter Hijma, Ceriel JH Jacobs, Rob V van Nieuwpoort and Henri E Bal. 'Cashmere: Heterogeneous Many-Core Computing'. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015, pp. 135–145.

[70] Seema Hiranandani, Ken Kennedy and Chau-Wen Tseng. 'Compiling Fortran D for MIMD distributed-memory machines'. In: *Communications of the ACM*, vol. 35, no. 8, 1992, pp. 66–80. ACM, 1992.

[71] M. Howison, E.W. Bethel and H. Childs. 'Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems'. In: *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, 2012, pp. 17–29. Washington, D.C., USA: IEEE, 2012. ISSN: 1077-2626.

[72] Andra-Ecaterina Hugo, Abdou Guermouche, Pierre-Andre Wacrenier and Raymond Namyst. 'Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach'. In: *Proc. IPDPSW'13 PhD Forum*. Washington, D.C., USA: IEEE, 2013, pp. 1050–1059. ISBN: 978-0-7695-4979-8.

[73] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[74] SA Kadir and Tazrian Khan. 'Parallel ray tracing using mpi and openmp'. In: *Project Report, Introduction to High Performance Computing, Royal Institute of Technology, Stockholm, Sweden*, 2008. 2008.

[75] SM Ashraful Kadir and Tazrian Khan. 'Parallel Ray Tracing using MPI: A Dynamic Load-balancing Approach'. In:

[76] Kamran Karimi, Neil G Dickson and Firas Hamze. 'A performance comparison of CUDA and OpenCL'. In: *arXiv preprint arXiv:1005.2581*, 2010. 2010.

[77] N.P. Karunadasa and D.N. Ranasinghe. 'Accelerating high performance applications with CUDA and MPI'. In: *ICIIS'2009*. Dec. 2009, pp. 331–336. DOI: 10.1109/ICIINFS.2009.5429842.

[78] Henry Kasim, Verdi March, Rita Zhang and Simon See. 'Survey on parallel programming model'. In: *IFIP International Conference on Network and Parallel Computing*. Springer. 2008, pp. 266–275.

[79] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo and Jaejin Lee. 'SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters'. In: *Proceedings of the 26th ACM international conference on Supercomputing*. ACM. 2012, pp. 341–352.

[80] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R Govindarajan, Albert Cohen and P Sadayappan. 'Compiler/runtime framework for dynamic dataflow parallelization of tiled programs'. In: *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, 2015, p. 61. ACM, 2015.

[81] Martin Kong, Louis-Noël Pouchet, P Sadayappan and Vivek Sarkar. 'PIPES: a language and compiler for task-based programming on distributed-memory clusters'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2016, p. 39.

[82] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić and Vivek Sarkar. 'HabaneroUPC++: A Compiler-free PGAS Library'. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 5.

[83] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann and Samuel Midkiff. 'A hybrid approach of OpenMP for clusters'. In: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, pp. 75–84.

[84] T. Liang, H. Li and J. Chiu. 'Enabling Mixed OpenMP/MPI Programming on Hybrid CPU/GPU Computing Architecture'. In: *Proc. IPDPSW'12, PhD Forum*. Washington, D.C., USA: IEEE, 2012, pp. 2369–2377. DOI: 10.1109/IPDPSW.2012.294.

[85] Kamal Lodaya and Pascal Weil. 'Series-parallel posets: algebra, automata and languages'. In: *STACS 98*. Springer. 1998, pp. 555–565.

[86] Thomas J McCabe. 'A complexity measure'. In: *Software Engineering, IEEE Transactions on*, vol. 4, 1976, pp. 308–320. IEEE, 1976.

[87] Sanyam Mehta, Gautham Beeraka and Pen-Chung Yew. 'Tile size selection revisited'. In: *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, 2013, p. 35. ACM, 2013.

[88] J Mellor-Crummey, V Adve, Bradley Broom, D Chavarría-Miranda, R Fowler, Guohua Jin, Ken Kennedy and Qing Yi. 'Advanced optimization strategies in the Rice dHPF compiler'. In: *Concurrency and Computation: Practice and Experience*, vol. 14, no. 8-9, 2002, pp. 741–767. Wiley Online Library, 2002.

[89] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. Tech. rep. 2012. DOI: www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[90] Ana Moreton, Arturo Gonzalez-Escribano and Diego R. Llanos. 'A Runtime Analysis for Communication Calculation'. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. poster, 2017.

[91] Ana Moreton-Fernandez and Arturo Gonzalez-Escribano. 'Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions'. In: *10th International Symposium on High-Level Parallel Programming and Applications (HLPP)*. Valladolid, Spain, 2017.

[92] Ana Moreton-Fernandez and Arturo Gonzalez-Escribano. 'Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions'. In: *Concurrency and Computation: Practice and Experience*, 2018. Wiley, 2018.

[93] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'A New High-Level Parallel Portable Language for Hierarchical Systems in Trasgo'. In: *Computational and Mathematical Methods in Science and Engineering (CMMSE)*. 2015.

[94] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'A Technique to Automatically Determine Ad-hoc Communication Patterns at Runtime'. In: *Parallel Computing*, 2017. North-Holland, 2017.

[95] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'Exploiting distributed and shared memory hierarchies with Hitmap'. In: *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE. 2014, pp. 278–286.

[96] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Multi-device Controllers: A library to Simplify Parallel Heterogeneous Programming'. In: *International Journal of Parallel Programming*, 2017, pp. 1–20. Springer, 2017.

[97] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R Llanos. 'On the run-time cost of distributed-memory communications generated using the polyhedral model'. In: *High Performance Computing & Simulation (HPCS), 2015 International Conference on*. IEEE. 2015, pp. 151–159.

[98] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Operators for Data Redistribution on the STL library and RayTracing Algorithm (Submitted)'. In: *Journal of Parallel and Distributed Computing*, 2018. Elsevier, 2018.

[99] Ana Moreton–Fernandez, Hector Ortega–Arranz and Arturo Gonzalez–Escribano. 'Controllers: An abstraction to ease the use of hardware accelerators'. In: *The International Journal of High Performance Computing Applications*, 2017. 2017. DOI: 10.1177/1094342017702962. eprint: http://dx.doi.org/10.1177/1094342017702962. URL: http://dx.doi.org/10.1177/1094342017702962.

[100] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano and Diego R Llanos. 'Supporting the Xeon Phi Coprocessor in a Heterogeneous Programming Model'. In: *European Conference on Parallel Processing*. Springer, Cham. 2017, pp. 457–469.

[101] Chris J Newburn, Serguei Dmitriev, Ravi Narayanaswamy, John Wiegert, Ravi Murty, Francisco Chinchilla, Rajiv Deodhar and Russ McGuire. 'Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor'. In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 1213–1225.

[102] NVIDIA. *NVIDIA CUDA C Programming Guide 7.5*. Last visit: June 14th, 2017. 2015. URL: %7Bhttp://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf%7D.

[103] CUDA Nvidia. 'Cublas library'. In: *NVIDIA Corporation, Santa Clara, California*, vol. 15, 2008, p. 27. 2008.

[104] Héctor Ortega Arranz. 'Parallel Approaches to Shortest-Path Problems for Multilevel Heterogeneous Computing'. PhD thesis. Universidad de Valladolid, 2015. URL: https://www.infor.uva.es/~hector/docs/phdthesis_ortega-arranz.pdf.

[105] Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Optimizing an APSP implementation for NVIDIA GPUs using kernel characterization criteria'. In: *The Journal of Supercomputing*, vol. 70, no. 2, 2014, pp. 786–798. Springer US, 2014. ISSN: 0920-8542. DOI: 10.1007/s11227-014-1212-z.

[106] Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano and Diego R. Llanos. 'TuCCompi: A Multi-layer Model for Distributed Heterogeneous Computing with Tuning Capabilities'. English. In: *International Journal of Parallel Programming*, vol. 43, no. 5, 2015, pp. 939–960. New York, NY, USA: Springer US, 2015. ISSN: 0885-7458. DOI: 10.1007/s10766-015-0349-6.

[107] Borja Pérez, José Luis Bosque and Ramón Beivide. 'Simplifying programming and load balancing of data parallel applications on heterogeneous systems'. In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM. 2016, pp. 42–51.

[108] Judit Planas, Rosa M Badia, Eduard Ayguadé and Jesus Labarta. 'Hierarchical task-based programming with StarSs'. In: *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, 2009, pp. 284–299. SAGE Publications Sage UK: London, England, 2009.

[109] L.-N. Pouchet, P. Zhang P. Sadayappan and J. Cong. 'Polyhedral-Based Data Reuse Optimization for Configurable Computing'. In: *ACM/SIGDA FPGA'13*. 2013, pp. 29–38.

[110] Louis-Noël Pouchet. 'Polybench: The polyhedral benchmark suite'. In: *URL: http://www. cs. ucla. edu/ ̃pouchet/software/polybench/[cited July,]* 2012. 2012.

[111] Benoit Pradelle, Philippe Clauss and Vincent Loechner. 'Adaptive runtime selection of parallel schedules in the polytope model'. In: *Proceedings of the 19th High Performance Computing Symposia*. Society for Computer Simulation International. 2011, pp. 81–88.

[112] David A Randall, Todd D Ringler, Ross P Heikes, Phil Jones and John Baumgardner. 'Climate modeling with spherical geodesic grids'. In: *Computing in Science and Engineering*, vol. 4, no. 5, 2002, pp. 32–41. 2002.

[113] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J Ramanujam, Atanas Rountev and P Sadayappan. 'Distributed memory code generation for mixed Irregular/Regular computations'. In: *Proc. PPoPP'2015*. ACM. 2015, pp. 65–75.

[114] *RayTrace code*. `http://www.purplealienplanet.com/node/20`. [Online; accessed 22-August-2016]. 2011.

[115] Chandan Reddy and Uday Bondhugula. 'Effective automatic computation placement and data allocation for parallelization of regular programs'. In: *Proceedings of the 28th ACM international conference on Supercomputing*. ACM. 2014, pp. 13–22.

[116] Ruymán Reyes and Francisco de Sande. 'Optimization strategies in different CUDA architectures using llCoMP'. In: *Microprocess. Microsyst.* Vol. 36, no. 2, Mar. 2012, pp. 78–87. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., Mar. 2012. ISSN: 0141-9331.

[117] Rochester Institute of Technology. *MPI implementation of hyperquicksort*. `https://www.cs.rit.edu/usr/local/pub/ncs/parallel/mpi/hqs.c`. [Online; accessed 22-August-2016]. 2003.

[118] Alberto Sanz, Rafael Asenjo, Juan Lopez, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi and Bradford L Chamberlain. 'Global data re-allocation via communication aggregation in Chapel'. In: *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE. 2012, pp. 235–242.

[119] Aroon Sharma, Darren Smith, Joshua Koehler, Rajeev Barua and Michael Ferguson. 'Affine loop optimization based on modulo unrolling in Chapel'. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 13.

[120] Thomas J Sheffler. *A portable MPI-based parallel vector template library*. Tech. rep. RIACS-TR-95.04. 1995.

[121] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou and Henk Sips. 'Workload partitioning for accelerating applications on heterogeneous platforms'. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, 2016, pp. 2766–2780. IEEE, 2016.

[122] Johannes Singler and Benjamin Konsik. 'The GNU libstdc++ parallel mode: software engineering considerations'. In: *Proceedings of the 1st international workshop on Multicore software engineering*. ACM. 2008, pp. 15–22.

[123] Johannes Singler, Peter Sanders and Felix Putze. 'MCSTL: The multi-core standard template library'. In: *Euro-Par 2007 Parallel Processing*. Springer, 2007, pp. 682–694.

[124] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Tech. rep. 95-11(R.1). 1995.

[125] Michel Steuwer and Sergei Gorlatch. 'SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems'. English. In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Vol. 7979. LNCS. Berlin, Germany: Springer Berlin Heidelberg, 2013, pp. 258–272. ISBN: 978-3-642-39957-2. DOI: 10.1007/978-3-642-39958-9_24.

[126] Tim Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[127] John E Stone, David Gohara and Guochun Shi. 'OpenCL: A parallel programming standard for heterogeneous computing systems'. In: *Computing in science & engineering*, vol. 12, no. 1-3, 2010, pp. 66–73. Institute of Electrical and Electronics Engineers, Inc.,| y USA United States, 2010.

[128] John A. Stratton, Sam S. Stone and Wen-Mei W. Hwu. 'MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs'. In: *LCPC'2008*. Ed. by José Nelson Amaral. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 16–30. ISBN: 978-3-540-89739-2.

[129] Zalán Szugyi, Márk Török and Norbert Pataki. 'Towards a Multicore C++ Standard Template Library'. In: *Proc. of Workshop on Generative Technologies (WGT 2011)*. 2011, pp. 38–48.

[130] TOP500.org. *Top500 Supercomputing Sites*. eng. WWW. on http://www.top500.org/. May 2017.

[131] Yuri Torres, Arturo Gonzalez-Escribano and Diego R. Llanos. 'uBench: exposing the impact of CUDA block geometry in terms of performance'. In: *The Journal of Supercomputing*, vol. 65, no. 3, 2013, pp. 1150–1163. Springer US, 2013. ISSN: 0920-8542. DOI: 10.1007/s11227-013-0921-z.

[132] Ramakrishna Upadrasta and Albert Cohen. 'Sub-polyhedral scheduling using (unit-) two-variable-per-inequality polyhedra'. In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 483–496.

[133] András Vajda. 'Multi-core and many-core processor architectures'. In: *Programming Many-Core Chips*. Springer, 2011, pp. 9–43.

[134] Arjan JC Van Gemund. 'The importance of synchronization structure in parallel program optimization'. In: *Proceedings of the 11th international conference on Supercomputing*. ACM. 1997, pp. 164–171.

[135] Sven Verdoolaege. 'Isl: An integer set library for the polyhedral model'. In: *Mathematical Software--ICMS 2010*. Springer, 2010, pp. 299–302.

[136] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Rubén Gran and Maŕia Garzarán. 'Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips'. In: *Procedia Computer Science*, vol. 51, 2015, pp. 140–149. Elsevier, 2015.

[137] Moisés Viñas, Zeki Bozkus and Basilio B Fraguela. 'Exploiting heterogeneous parallelism with the Heterogeneous Programming Library'. In: *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, 2013, pp. 1627–1638. Elsevier, 2013.

[138] Bruce Wagar. 'Hyperquicksort: A fast sorting algorithm for hypercubes'. In: *Hypercube Multiprocessors*, vol. 1987, 1987, pp. 292–299. SIAM, 1987.

[139] Sandra Wienke, Paul Springer, Christian Terboven and Dieter an Mey. 'OpenACC—first experiences with real-world applications'. In: *European Conference on Parallel Processing*. Springer. 2012, pp. 859–870.

[140] Marc H Willebeek-LeMair and Anthony P. Reeves. 'Strategies for dynamic load balancing on highly parallel computers'. In: *IEEE Transactions on parallel and distributed systems*, vol. 4, no. 9, 1993, pp. 979–993. IEEE, 1993.

[141] Chao-Tung Yang, Chih-Lin Huang and Cheng-Fang Lin. 'Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters'. In: *Computer Physics Communications*, vol. 182, no. 1, 2011, pp. 266–269. Elsevier, 2011.

[142] C.T. Yang, C.L. Huang and C.F. Lin. 'Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters'. In: *Computer Physics Communications*, vol. 182, no. 1, 2011, pp. 266–269. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., 2011. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2010.06.035.

[143] Tomofumi Yuki and Sanjay Rajopadhye. *Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs*. Tech. rep. CS13-105. June 2013.