Universidad de Valladolid

Escuela Técnica Superior de
Ingeniería Informática

Departamento de Informática

Tesis Doctoral:

Supporting general data structures and execution models in runtime environments

Presentada por Javier Fresno Bausela para optar al grado de doctor por la Universidad de Valladolid

Dirigida por:
Dr. Arturo Gonzalez Escribano

Valladolid, Julio 2015

Resumen

Durante las últimas décadas, los sistemas paralelos se han vuelto cada vez más populares tanto en el ámbito de la computación de altas prestaciones como en el de la computación convencional. Esto se debe a un enorme incremento en las capacidades de las plataformas paralelas. Para aprovechar las ventajas de estas plataformas, se necesitan nuevas herramientas de programación. En primer lugar, se necesitan modelos de programación con abstracciones de alto nivel para poder representar apropiadamente los algoritmos paralelos. Además, los entornos paralelos que implementan dichos modelos requieren sistemas en tiempo de ejecución completos que ofrezcan diferentes paradigmas de computación a los programadores. De este modo es posible construir programas que puedan ser ejecutados eficientemente en diferentes plataformas.

Existen diferentes áreas a estudiar con el fin de construir un sistema en tiempo de ejecución completo para un entorno paralelo. Esta Tesis aborda dos problemas comunes: el soporte unificado de datos densos y dispersos, y la integración de paralelismo orientado a mapeo de datos y paralelismo orientado a flujo de datos.

En primer lugar, un entorno paralelo genérico requiere integrar manejo para datos densos y dispersos usando un interfaz común siempre que sea posible. Nosotros proponemos una solución que desacopla la representación, partición y reparto de datos, del algoritmo y de la estrategia de diseño paralelo.

En segundo lugar, un entorno paralelo de estás características debe permitir programar aplicaciones dinámicas y de flujo de datos, las cuales son difíciles de realizar con las herramientas actualmente disponibles. En esta Tesis, introducimos un nuevo modelo de programación basado en el paradigma de flujo de datos, donde diferentes actividades pueden ser arbitrariamente enlazadas para formar redes genéricas pero estructuradas que representan el cómputo global.

Keywords

Computación paralela, Entornos paralelos, Manejo de datos dispersos, Modelos de flujo de dato, Patrones paralelos, Computación dinámica

Abstract

Parallel computing systems have become increasingly popular over the past decades in both high performance and mainstream computing. The reason is that there has been a huge increase in the capabilities of parallel platforms. To take advantage of these platforms, we require new programming tools. First, models with high-level abstractions to represent appropriately parallel algorithms. Second, the programming frameworks that implement these models need complete runtime systems that offer different parallel paradigms to the framework programmers. This way, it is possible to build programs that can be efficiently executed in different platforms.

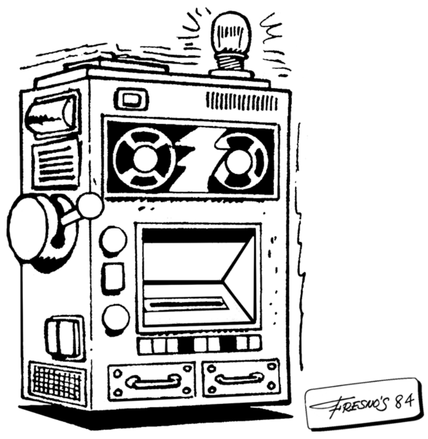
There are different areas to study in order to develop a complete runtime system for a parallel framework. This Ph.D. Thesis addresses two common problems: The unified support for dense and sparse data, and the integration of data-mapping and data-flow parallelism.

In the first place, a generic parallel framework requires to integrate dense and sparse data management using a common interface whenever possible. We propose a solution that decouples data representation, partitioning, and layout from the algorithmic and from the parallel strategy decisions of the programmer.

In the second place, such a parallel framework should allow to program dynamic and dataflow applications, which is a challenging task with current available tools. In this Thesis, we introduce a new programming model based on the dataflow paradigm, where different activities can be arbitrarily linked, forming generic but structured networks that represent the overall computation.

Keywords

Parallel programming, Parallel frameworks, Sparse data management, Dataflow models, Parallel patterns, Dynamic computation.


FRESNO'S 84

Agradecimientos

Ya está, es el fin de esta aventura. Cuando empecé, hace ya unos años, poco me imaginaba qué era hacer una tesis, cómo sería la vida del investigador, qué cosas nuevas aprendería, a qué lugares me llevaría y qué gente nueva conocería. Ha sido un camino largo y lleno de altibajos que ha sido posible acabar gracias a la ayuda de mucha gente. Ahora quiero dedicar unas lineas a quien me ha acompañado en este viaje.

En primer lugar quiero agradecer a Arturo y Diego por todo la ayuda recibida. Han sido mucho más que mis tutores y gracias a su trabajo he podido completar la Tesis. Quiero agradecer a mis compañeros Yuri, Sergio, Héctor, Álvaro y Ana, ellos han hecho que todo el esfuerzo valiese la pena. No ha habido problema al que no me pudiera enfrentar gracias a las bromas, los juegos y los coffee-breaks que hemos compartido.

También quiero nombrar a mis padres y a mi hermano Andrés que han sido un gran apoyo. Por último aunque no por eso menos tengo que dar las gracias a Cristina por estar siempre a mi lado y por darme los ánimos que necesitaba para acabar este trabajo.

Javier Fresno Bausela

This research is partly supported by the Junta de Castilla y León (ATLAS project, VA172A12-2); the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF); MICINN (Spain) and ERDF program of the European Union (MOGECOPP project TIN2011-25639, HomProg-HetSys project TIN2014-58876-P, CAPAP-H5 network TIN2014-53522-REDT).

Contents

List of Figures

List of Tables

List of Listings

Resumen de la tesis

La computación paralela se ha vuelto más popular durante las últimas décadas tanto en el ámbito de la computación de alto rendimiento como en el de la computación convencional. Las mejoras en el proceso de fabricación de procesadores han permitido crear una nueva generación de chips multi-núcleo, supercomputadores masivamente paralelos y aceleradores.

Un problema recurrente para los desarrolladores de programas paralelos es sacar provecho de las capacidades de estos nuevos sistemas. Los modelos de hilos y paso de mensajes han sido, durante mucho años, las soluciones más exitosas. Sin embargo, ofrecen un nivel de abstracción muy bajo que no es adecuado para los retos de las nuevas generaciones de hardware.

Se necesitan nuevos modelos de programación con abstracciones de alto nivel adecuadas para representar los algoritmos paralelos para estos nuevos sistemas. Los entornos de programación que usen estos modelos de alto nivel permitirán a los programadores concentrarse en las decisiones de diseño, mientras que los sistemas en tiempo de ejecución se encargarán de hacer las complicadas optimizaciones en función de la arquitectura. Potenciando de esta forma tanto el rendimiento como la portabilidad de los programas.

R.1 Motivación

Esta sección presenta la motivación de la Tesis, se discute la evolución de los sistemas paralelos y la necesidad de nuevos modelos de programación junto con sistemas en tiempo de ejecución para controlarlos. También se introduce el trabajo del grupo de investigación Trasgo en este tema particular y los retos que esta Tesis intenta resolver.

R.1.1 La evolución de los sistemas paralelos

El paralelismo es una área importante dentro de la computación. Muchos problemas son tan grandes que no puede ser resueltos de forma secuencial en un tiempo razonable. La necesidad de proporcionar soluciones a estos problemas es lo que ha fomentado el desarrollo de los sistemas paralelos. Ha habido una gran cantidad de cambios en los sistemas paralelos durante las últimas décadas, con un incremento constante del número de elementos de proceso, rendimiento y capacidad de memoria.

La historia de la computación paralela se remonta a los años 60 con la creación de los primeros supercomputadores. Los supercomputadores pioneros tenía una capacidad de unos pocos megaflop/s. Desde entonces, la evolución de la tecnología ha hecho posible actualizaciones en el hardware que han mejorado año tras año. Esto ha sido registrado por el proyecto Top500 que clasifica los 500 sistemas de mayor potencia del mundo. Desde el inicio del proyecto en 1993, el rendimiento del sistema en la primera posición de la lista ha mostrado un crecimiento constante. La generación actual de supercomputadores son máquinas masivamente paralelas con millones de unidades de computo que han alcanzado la marca de los petaflop/s. Se espera que la marca de exaflop/s será alcanzada en menos de diez años.

Desde que en el año 2007 Nvidia lanzase la tarjeta GPU (*Graphics Processing Unit*) Tesla diseñada para el paradigma de programación GPGPU (*General-Purpose computing on GPUs*), el uso de co-procesadores junto con procesadores de propósito general se ha convertido en una gran tendencia en el mundo de la supercomputación. Muchos de los últimos supercomputadores están basados en aceleradores, con 75 sistemas en la última lista del Top500 (noviembre de 2014). Estos aceleradores tienen un hardware especializado que los hacen adecuados por estilos particulares de computación. Por tanto pueden ser usados para acelerar partes específicas de las aplicaciones. El ejemplo de tecnología más exitosa son las tarjetas GPU Nvidia con el modelo de programación CUDA. Otro ejemplo son las tarjetas Intel Xeon Phi, lanzadas en el año 2012. Son aceleradores con cores basados en tecnología x86 con unidades vectoriales y un ancho de banda mejorado. Pueden ser programadas tanto como co-procesador como usando herramientas para programación distribuida.

Sin embardo, la computación paralela no solo es relevante en la comunidad de alto rendimiento. También se ha hecho hueco en la computación convencional. Hasta los años 2000, el incremento en la frecuencia de los procesadores era la fuente predominante de mejora en el rendimiento de los ordenadores. Durante esa época, los fabricantes de procesadores alcanzaron un límite de frecuencia. No era posible aumentar el rendimiento de los equipos simplemente aumentando la frecuencia de funcionamiento de los procesadores sin incrementar el calor generado dando lugar a su vez a un incremento del consumo de energía e introduciendo problemas de disipación. Los fabricantes empezaron a añadir más transistores en los chips para incrementar la lógica dando lugar a la tecnología multi-núcleo. Actualmente, existe una gran proliferación de procesadores multi-núcleo en el mercado de masas. Procesadores domésticos con hasta 8 núcleos son fáciles de encontrar incluso para equipos móviles como smartphones.

R.1.2 Limitaciones de los modelos actuales de computación paralela

La programación paralela se considera difícil en la práctica. Existen numerosos modelos de programación paralela que resuelven problemas particulares. Las bibliotecas y herramientas paralelas más usadas están basadas en modelos de paso de mensajes para memoria distribuida, modelos de hilos para memoria compartida y modelos basados en kernels para aceleradores. Todos ellos ofrecen un nivel bajo de abstracción que fuerza a los programadores a tratar de forma manual con la sincronización, con la localidad de datos, y/o con la distribución de carga.

Los enormes cambios en las arquitecturas han hecho que el desarrollo de los programas paralelos sea incluso más difícil. Programar clusters de equipos multi-núcleo requiere combinar diferentes herramientas con diferentes modelos de programación. Además, la aparición de los aceleradores ha añadido una nueva capa de complejidad. Un programador debe ser experto en MPI, OpenMP y CUDA (o en tecnologías equivalentes) para ser capaz de aprovechar la actual generación de sistemas paralelos. Mientras que cada tecnología tenga una aproximación diferente a la tarea de programación paralela, el diseño e implementación de soluciones para cada modelo de programación será muy diferente. Además, la variedad de maquinas heterogéneas hace casi imposible producir códigos portables debido a las optimizaciones necesarias para aprovechar todo el rendimiento potencial de cada tipo de hardware.

Necesitamos entornos de programación con nuevos y unificados modelos de programación. Estos modelos deben ofrecen abstracciones de alto nivel para poder representar adecuadamente los algoritmos paralelos. Esto permitirá a los programadores centrarse en las decisiones de diseño. Confiando para ello en los compiladores para realizar las complejas optimizaciones usando sistemas en tiempo de ejecución eficientes y adaptables.

R.1.3 Trabajo llevado a cabo en el grupo de investigación Trasgo

El grupo de investigación Trasgo [67] está desarrollando un entorno de programación modular denominado Trasgo. Su modelo de programación está basado en especificaciones de paralelismo anidado de alto nivel. Su tecnología de generación de código pretende obtener programas paralelos completos para cluster heterogéneos ayudándose de una biblioteca en tiempo de ejecución llamada Hitmap. Esta biblioteca implementa funciones para crear, manipular, distribuir y comunicar jerarquías de arrays de forma eficiente. Trasgo transforma el código paralelo de alto nivel proporcionado por el programador en código fuente con llamadas a la biblioteca Hitmap. Trasgo se encarga de la asignación de datos y de las trasformaciones de optimización. El resultado final es un código altamente optimizado y además portable. Los siguiente párrafos presentan el modelo Trasgo con más detalle.

El entorno de programación Trasgo

Trasgo [58] es un entorno de programación basado en especificaciones paralelas anidadas que permiten expresar de forma sencilla combinaciones complejas de paralelismo de datos y tareas usando un esquema común. Una de las características más importantes es que su modelo oculta los detalles de asignación de datos y planificación.

El entorno de programación Trasgo soporta su propio modelo de programación, basado en un modelo de proceso de álgebra sencillo. Trasgo proporciona un lenguaje paralelo de alto nivel. Las características principales de este nuevo lenguaje de programación son: (1) es un lenguaje paralelo anidado de coordinación; (2) usa interfaces extendidas de funciones para permitir al compilador detectar flujo de datos; y (3) usa primitivas paralelas abstractas y unificadas para declarar tareas lógicas de grano grueso o grano fijo.

El entorno de programación incluye un sistema de módulos para generar la información sobre partición y asignación de datos que es consultado en tiempo de ejecución para crear códigos de comunicación abstractos.

La biblioteca Hitmap

La biblioteca Hitmap [59] proporciona soporte en tiempo de ejecución a Trasgo. Es una biblioteca con funcionalidades de *tiling* jerárquico. Está diseñada para simplificar el uso de una vista global o local de la computación paralela, permitiendo la creación, manipulación, distribución y comunicación eficiente de tiles y sus jerarquías. En Hitmap, las técnicas de partición y balanceo de datos son elementos independientes que pertenecen a un sistema de módulos. Los módulos son invocados desde el código y aplicados en tiempo de ejecución según se necesitan para distribuir los datos usando la información interna de la topología del sistema. El programador no tiene que razonar en términos del número de procesadores físicos. Por el contrario, el programador puede usar patrones de comunicación abstractos para distribuir tiles. Por tanto, es sencillo realizar las operaciones de programación y depuración con estructuras completas de datos. La biblioteca Hitmap soporta funcionalidades para: (1) generar una estructura de topología virtual; (2) realizar las asignaciones de datos a los diferentes procesos usando técnicas de balanceo de carga; (3) determinar de forma automática los procesadores inactivos en cualquier etapa de la computación; (4) identificar los procesadores vecinos para usar en las comunicaciones; y (5) construir patrones de comunicación para ser reutilizados en las diferentes iteraciones del algoritmo.

R.1.4 Avances hacia un modelo de programación unificado

Soportar todas las abstracciones de un modelo de programación unificado requiere un sistema en tiempo de ejecución con un alcance más amplio que el que ofrece la biblioteca Hitmap. Al inicio del trabajo de investigación, la biblioteca Hitmap solo tenía funcionalidades para soportar mapeo estático en arrays densos. Necesitábamos reemplazar o extender la biblioteca Hitmap para soportar otras clases de aplicaciones paralelas u otros paradigmas de programación. Es en ese punto en el que el objetivo de esta tesis surgió. Esta tesis analiza dos de las limitaciones más importantes de los sistemas en tiempo de ejecución actuales, como Hitmap:

- *Soporte unificado para datos densos y dispersos.*

  No muchos entornos de programación paralela integran de manera transparente soporte para estructuras de datos densos y dispersos. La mayoría de los lenguajes de programación paralela sólo tienen soporte nativo para matrices densas, incluyendo primitivas y herramientas para hacer frente a la localidad de datos y/o distribución para estructuras de datos densos. Programar aplicaciones orientadas a datos dispersos implica gestionar los datos manualmente con un esfuerzo de programación elevado o utilizar bibliotecas específicas que pueden no seguir el mismo enfoque conceptual. En ambos casos, la reutilización del código desarrollado previamente para estructuras de datos densos es muy pobre.

- *Integración de paradigmas y modelos paralelos dinámicos.*

  Estructuras paralelas estáticas simples son fáciles de programar con las soluciones de programación paralela actuales. Sin embargo, usarlos para programar aplicaciones dinámicas de flujo de datos, especialmente en las plataformas híbridas de memoria distribuida y compartida, es un reto importante. Esto es debido a que las soluciones actuales ofrecen un nivel de abstracción demasiado bajo para hacer frente los problemas de sincronización más complejos. Trabajar con combinaciones jerárquicas de paradigmas paralelos conduce a códigos de alta complejidad, con muchas decisiones sobre el mapping codificadas de forma explícita en el código.

Trabajo previo realizado en el grupo ya había abordado el soporte de dispositivos heterogéneos en la biblioteca Hitmap [124]. La tesis *Hierarchical Transparent Programming for Heterogeneous Computing* [123] presenta una serie de políticas para seleccionar parámetros de ejecución adecuados para explotar la arquitectura de los dispositivos GPU de manera eficiente. Esa tesis incluye un estudio de nuevos niveles de particionado y sus políticas asociadas, transmisión de datos a través de dispositivos GPU, subdivisión de tareas según las capacidades del dispositivo, y selección de la forma y tamaño apropiado para conjuntos de hilos.

R.2 Pregunta de investigación

En función de los problemas identificados en el apartado anterior, la pregunta de investigación que se resuelve en esta Tesis Doctoral es la siguiente:

> *Es posible crear un sistema en tiempo de ejecución para un lenguaje de programación genérico que ofrezca (1) abstracciones comunes para manejo de datos densos y dispersos, y (2) soporte para paralelismo orientado a mapeo y flujo de datos para entornos híbridos de memoria compartida y distribuida?*

Para poder responder a esta pregunta de investigación hemos realizado las tareas que se describen a continuación:

1. Hemos propuesto una clasificación de las herramientas paralelas en dos categorías, orientadas a mapeo de datos u orientadas a flujo de datos, en función de su modelo de ejecución.

2. Hemos llevado a cabo un análisis de diferentes estructuras de datos, algoritmos, bibliotecas, y entornos de programación para la gestión de datos dispersos.

3. Hemos ampliado la metodología de programación de la biblioteca Hitmap con una etapa adicional para seleccionar el tipo de estructura de datos. La nueva metodología integra conceptualmente dominios densos y dispersos.

4. Hemos rediseñado la biblioteca Hitmap siguiendo la metodología propuesta, añadiendo varias representaciones para grafos y matrices dispersas. La nueva biblioteca integra soporte denso y disperso con las mismas abstracciones. Diferentes programas de prueba se han implementado para esta versión y se ha llevado a cabo trabajo experimental para validar la solución en términos de rendimiento y de facilidad de programación.

5. Hemos llevado a cabo un estudio de lenguajes de modelado formales para la descripción de sistemas paralelos. Este tipo de lenguajes representan las dependencias de programas paralelos en una manera formal, tienen una semántica de ejecución bien definida y teorías matemáticas adecuadas para análisis de procesos.

6. Hemos desarrollado un modelo teórico basado en mecanismos de flujo de datos, utilizando como base un lenguaje de modelado formal. El uso de un modelo establecido añade propiedades interesantes como analizabilidad. Por lo tanto, muchas de las propiedades de los sistemas, tales como los interbloqueos, se pueden detectar automáticamente.

7. Hemos diseñado e implementado un prototipo para el modelo introducido anteriormente. El prototipo propone una forma genérica para describir de programas en forma de red reconfigurable de actividades y contenedores de datos que son arbitrariamente interconectados.

R.3 Resumen de contribuciones

La biblioteca Hitmap ha sido usada en esta Tesis como base para crear un sistema en tiempo de ejecución para un entorno de programación genérico. Está sección resume las contribuciones de la Tesis y las publicaciones producidas. Las contribuciones incluyen el trabajo previo necesario para estudiar las técnicas automáticas de la biblioteca Hitmap y el trabajo principal para resolver los dos problemas identificados que aparecen en los sistemas paralelos actuales.

R.3.1 Trabajo previo

Se ha realizado cierto trabajo previo a fin de comprender las capacidades de la biblioteca Hitmap original, y para determinar sus carencias. Hemos estudiado las técnicas automáticas de partición de datos aplicadas a los métodos multinivel. En particular, se ha estudiado el benchmark NAS MG [12]. Para este problema, hemos desarrollado una aplicación Hitmap usando el sistema de módulos para la partición y distribución automática de datos. El trabajo ha dado lugar a la publicación de los siguientes documentos:

1. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Automatic Data Partitioning Applied to Multigrid PDE Solvers'. En: *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*. February 9-11, Ayia Napa, Cyprus: IEEE, feb. de 2011, págs. 239-246. ISBN: 978-1-4244-9682-2. DOI: 10.1109/PDP.2011.38

2. Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno y Diego R. Llanos. 'An Extensible System for Multilevel Automatic Data Partition and Mapping'. En: *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, n.º 5, mayo de 2014, págs. 1145-1154. Mayo de 2014. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.83

R.3.2 Soporte unificado para datos densos y dispersos

Una de las limitaciones de muchos sistemas paralelos es que no proporcionan soporte unificado para estructuras de datos densos y dispersos. Nuestro objetivo era integrar ese soporte en la biblioteca Hitmap. Hemos ampliado la metodología de programación de Hitmap para integrar conceptualmente dominios densos y dispersos. Además, hemos desarrollado una nueva versión de la biblioteca siguiendo la metodología propuesta. La nueva implementación permite utilizar varias estructuras para grafos y matrices dispersas con las mismas abstracciones para estructuras densas. Estas contribuciones han sido publicadas en los siguientes documentos:

3. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Integrating dense and sparse data partitioning'. En: *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*. Ed. por J. Vigo-Aguiar. June 26-29, Alicante, Spain, 2011, págs. 532-543

4. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Extending a hierarchical tiling arrays library to support sparse data partitioning'. En: *The Journal of Supercomputing*, vol. 64, n.º 1, 2013, págs. 59-68. Springer US, 2013. ISSN: 0920-8542. DOI: 10.1007/s11227-012-0757-y

5. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Data abstractions for portable parallel codes'. En: *Proceedings of the Ninth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Ed. por HiPEAC. 14-20 July, Fiuggi, Italy, 2013, págs. 85-88

6. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Blending Extensibility and Performance in Dense and Sparse Parallel Data Management'. En: *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, n.º 10, 2014, págs. 2509-2519. IEEE, 2014. DOI: 10.1109/TPDS.2013.248

R.3.3 Integración de paradigmas y modelos paralelos con estructuras paralelas dinámicas

El segundo problema abordado en la tesis es el soporte de paradigmas y modelos con estructuras paralelas dinámicas. Estructuras paralelas estáticas son fáciles de programar utilizando herramientas comunes de programación paralela. Sin embargo, la programación de aplicaciones dinámicas y de flujo de datos es más difícil. Algunas soluciones disponibles se centran en un paradigma paralelo concreto, o sólo se pueden utilizar en plataformas particulares. Un sistema en tiempo de ejecución ideal para un sistema de programación genérico debe dar soporte a estos paradigmas y modelos dinámicos paralelos, así también como a las estructuras de comunicación estáticas clásicas.

Hemos estudiado las redes de Petri [95] y las redes de Petri coloreadas [77] como lenguajes de modelado para la descripción de sistemas paralelos. Usando estos lenguajes como base, hemos desarrollado un modelo teórico de flujo de datos. Hemos ampliado Hitmap para soportar el modelo propuesto. Esta versión de Hitmap nos permite describir programas como redes reconfigurables de actividades y contenedores de datos arbitrariamente interconectados. El trabajo realizado ha sido publicado en los siguientes documentos:

7. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Exploiting parallel skeletons in an all-purpose parallel programming system'. En: *Science and Supercomputing in Europe - research highlights (HPC-Europa2 project)*, 2012. 2012

8. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. 'Runtime Support for Dynamic Skeletons Implementation'. En: *Proceedings of the 19th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. July 22-25, Las Vegas, NV, USA., 2013, págs. 320-326

9. Javier Fresno, Arturo Gonzalez-Escribano y Diego R. Llanos. *Dataflow Programming Model for Hybrid Distributed and Shared Memory Systems*. Manuscript submitted for publication. 2015

R.4 Respuesta a la pregunta de investigación

El trabajo presentado en esta tesis nos permite concluir que la pregunta de investigación tiene una respuesta afirmativa. La biblioteca Hitmap estaba diseñada a ser utilizada como un sistema en tiempo de ejecución para el sistema de computación paralela Trasgo. Sin embargo, carecía de soporte para datos dispersos y para estructuras dinámicas basadas en flujo de datos. Con el trabajo de investigación realizado en esta Tesis, hemos sido capaces de extender la biblioteca para eliminar estas limitaciones.

Chapter 1

Introduction

Parallel computing has become increasingly popular over the past decades in high performance computing. Nowadays, it has also reached the mainstream computing communities. Improvements in processor manufacturing have brought a new generation of many-core chips, massively parallel supercomputers, and accelerator systems. Moreover, there has been a widespread deployment of multi-core processors in commodity desktop, laptop, and mobile computers.

A recurrent problem for the parallel developers has been to harness the capabilities of these new systems. Threads and message passing have been, for many years, the most successful parallel programming models. However, they offer low-level abstractions that are not suitable for the challenges of the new generation of hardware and memory architectures.

New programming models with higher-level abstractions are needed to represent parallel algorithms for these new systems. Frameworks using these high-level models would allow the programmers to focus on the design decisions, relaying on the runtime system to make the associated complex mapping optimizations in terms of target architecture and runtime situations, leveraging both performance and portability.

1.1 Motivation

This section presents the motivation of the Thesis, discussing the evolution of parallel systems and the need of 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 The evolution of parallel systems

Parallelism is an important computing topic. Many computing problems are so huge that they cannot be solved sequentially in a reasonable time. The need to provide a solution to these kind of problems has pushed forward the development of parallel systems. There have been impressive changes in the parallel computing systems during the last decades, with a constant increase in the number of processing elements, performance, and memory capacity. Figure 1.1 contains a timeline that we have made showing the evolution of parallel computing systems, according to four different categories: Supercomputers, accelerators, mainstream computers, and mobile computing. Some of the most important events are highlighted. The sources of the timeline are [9, 22, 35, 36, 37, 99, 121, 132, 133, 134].

Parallel computing history goes back to the 1960s, when the first supercomputers were introduced. Therefore, it has been usually associated with high performance computing. The pioneer supercomputers were capable of only a few megaflop/s. Since then, the evolution of technology has made possible hardware upgrades that got better and better which each passing year. This has been recorded by the Top500 project, which ranks the 500 most powerful computer systems in the world. Since the start of the project in 1993, the performance of the top system in the list has always shown a steadily grown. Current supercomputer generation, massive parallel machines with millions of processing units, have reached the 10 petaflop/s barrier. It is expected that the exascale mark will be beaten in less than 10 years.

Since Nvidia launched its Tesla GPU (*Graphics Processing Unit*) card designed for the GPGPU paradigm (*General-Purpose computing on GPUs*) in 2007, the use of co-processors along with the general-purpose processors has become a big trend in supercomputing. Many of the latest supercomputers are based on accelerators, with 75 accelerated systems in the last Top500 list (November 2014). These accelerators have a specialized hardware that make then suitable for particular types of computation. Thus, they can be used to speed up specific parts of applications. The most successful example of this technology is the Nvidia GPU cards with the CUDA programming model. Another example is Intel Xeon Phi, released in 2012. It is an accelerator with many 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.

However, parallel computing is not only relevant in the high performance community. It has also become important for mainstream computing community. Until the 2000s, frequency scaling was the dominant source of improvement in computer performance.

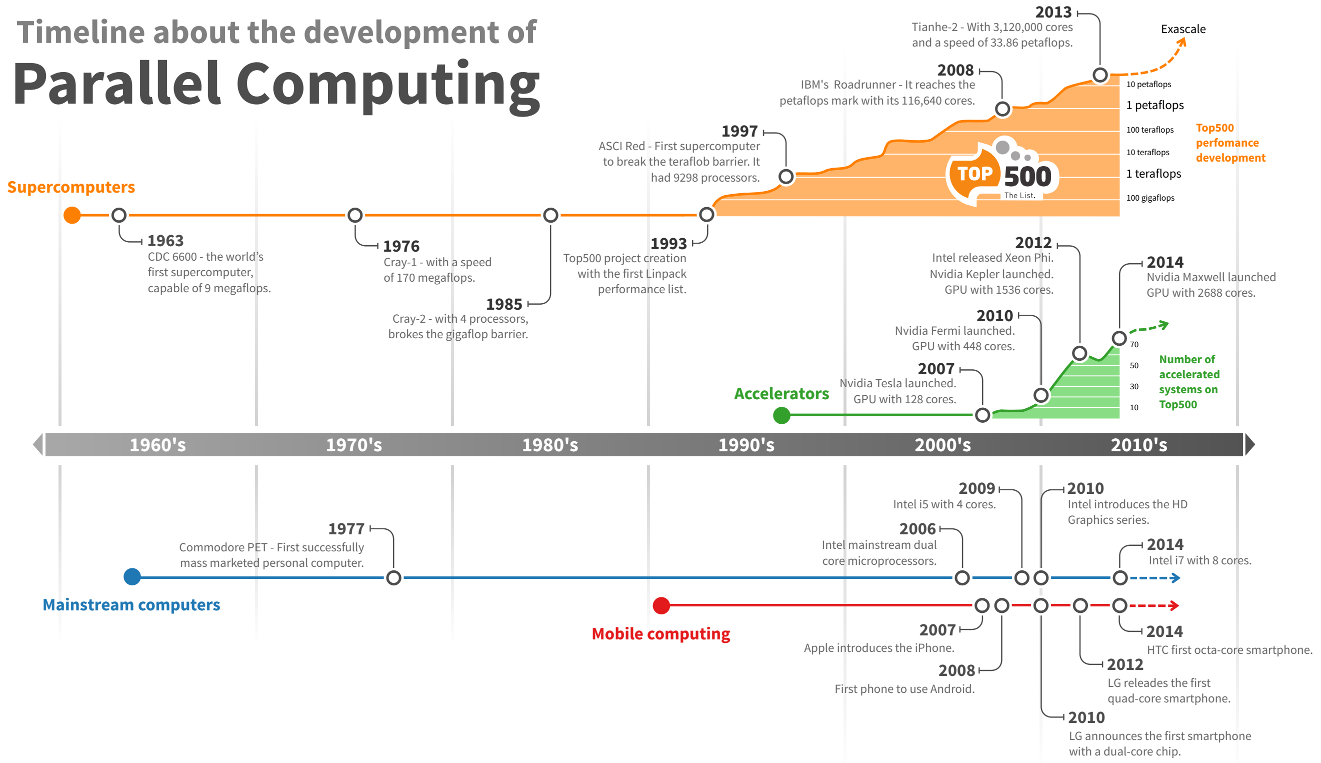

Timeline about the development of
Parallel Computing
Supercomputers
1963
CDC 6600 - the world's first supercomputer, capable of 9 megaflops.
1976
Cray-1 - with a speed of 170 megaflops.
1985
Cray-2 - with 4 processors, brokes the gigaflop barrier.
1993
Top500 project creation with the first Linpack performance list.
1997
ASCI Red - First supercomputer to break the teraflob barrier. It had 9298 processors.
2008
IBM's Roadrunner - It reaches the petaflops mark with its 116,640 cores.
2013
Tianhe-2 - With 3,120,000 cores and a speed of 33.86 petaflops.
Exascale
10 petaflops
1 petaflops
100 teraflops
10 teraflops
1 teraflops
100 gigaflops
TOP 500
The List.
Top500 perfomance development
Accelerators
2007
Nvidia Tesla launched. GPU with 128 cores.
2010
Nvidia Fermi launched. GPU with 448 cores.
2012
Intel released Xeon Phi. Nvidia Kepler launched. GPU with 1536 cores.
2014
Nvidia Maxwell launched GPU with 2688 cores.
70
50
30
10
Number of accelerated systems on Top500
1960's
1970's
1980's
1990's
2000's
2010's
Mainstream computers
1977
Commodore PET - First successfully mass marketed personal computer.
2006
Intel mainstream dual core microprocessors.
2009
Intel i5 with 4 cores.
2010
Intel introduces the HD Graphics series.
2014
Intel i7 with 8 cores.
Mobile computing
2007
Apple introduces the iPhone.
2008
First phone to use Android.
2010
LG announces the first smartphone with a dual-core chip.
2012
LG releades the first quad-core smartphone.
2014
HTC first octa-core smartphone.

Figure 1.1: Parallel computing timeline.

Around that time, processor vendors reached a frequency limit. There was just not possible to improve processor performance by increasing clock speeds without an increase of generated heat, leading to excessive power consumption, and dissipation problems. Vendors started adding more transistors in the chips to increase the logic, giving way to the multi-core technology. Nowadays, there is a huge proliferation of multi-core computers into the mass market. Commodity multi-core processors with up to 8 cores are easy to find, even for mobile devices such as smartphones.

1.1.2 Limitations of current parallel programming models

Parallel programming is considered hard in practice. There are many programming models that solve particular problems. Figure 1.2 shows the evolution of the scholarly literature in the parallel computing field. We have obtained this information from the Google Scholar project [74], a search engine for different types of academic works such as papers, theses, books, or technical reports. The figure shows that the most used parallel programming libraries and tools are based on message-passing models for distributed-memory, thread models for shared-memory environments, or kernel-based models for accelerators. All of them offer a low-level abstraction, forcing the programmers to manually deal with the synchronization, data locality, and/or distribution.

The huge changes in the architectures had made development of parallel programs even more difficult. Programming for multi-core clusters requires to mix several tools with different programming models. Moreover, the appearance of accelerators has added an additional layer of complexity. A programmer must be proficient in MPI, OpenMP, and CUDA (or equivalent technologies) to be able to take advantage of the current generation of parallel systems. As long as each technology has a different approach to the parallel programming task, the design and implementation of solutions for each programming model will be quite different. Moreover, the variety of heterogeneous machines makes almost impossible to produce portable codes due to the required optimizations needed to leverage their performance potential.

We need frameworks with new, unified parallel models. These models should offer high-level abstractions to appropriately represent parallel algorithms. This would allow the programmers to focus on the design decisions. They should rely on the compilers to do the associated complex optimizations, using highly-efficient and adaptable runtime systems.

1.1.3 Work carried out by Trasgo group

The Trasgo research group [67] is developing a modular programming framework called Trasgo. Its programming model is based on high-level, nested-parallel specifications. Its code generation technology aims to generate complete parallel programs for heterogeneous clusters relying on a runtime library called Hitmap. This library implements functions to efficiently create, manipulate, map, and communicate hierarchical tiling arrays. The high-level

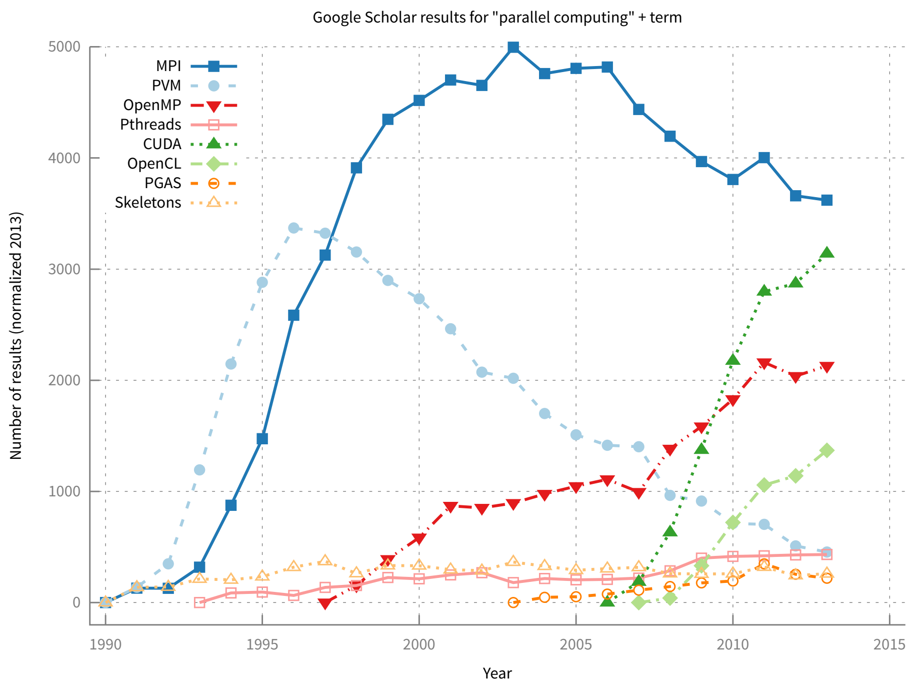


Figure 1.2: Comparison of the number of publications about parallel computing in the literature.

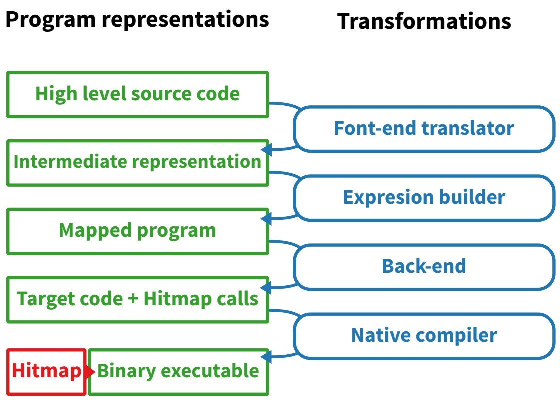


Figure 1.3: Trasgo framework transformations.

parallel code provided by the programmer is transformed by Trasgo, which takes care of the mapping and optimization transformations, into a source code augmented with Hitmap calls. The final result is a highly-optimized but portable application. The following paragraphs present the Trasgo model in more detail.

The Trasgo programming framework

Trasgo [58] is a programming framework based on high-level and nested-parallel specifications that allows to easily express several complex combinations of data and task parallelism within a common scheme. One of the most important features is that its model hides the data-layout and scheduling details.

The Trasgo programming framework supports its own programming model, based on a simple process-algebra model and exploiting data-distribution algebras. Trasgo provides a high-level parallel language. The main features of this new language are the following: (1) It is a nested parallel coordination language; (2) it uses extended function interfaces to allow the compiler to detect data-flow; and (3) it uses an abstract and unified parallel primitive for coarse- or fine-grain logical tasks.

The framework includes a plug-in system with several modules that generate partition and mapping information, which may be queried at runtime to create abstract communication codes independent of the mapping details. The framework uses a back-end to translate the final internal representation to a parallel source code. Figure 1.3 shows the different transformations used in Trasgo to produce the final binary program starting from the high level source code.

The Hitmap library

The Hitmap library [59] provides runtime support to Trasgo. It is a highly-efficient library for hierarchical tiling and mapping of arrays. It is designed to simplify the use of a global or local view of the parallel computation, allowing the creation, manipulation, distribution, and efficient communication of tiles and tile hierarchies. In Hitmap, data-layout and load-balancing techniques are independent modules that belong to a plug-in system. The techniques are invoked from the code and applied at runtime when needed, using internal information about the target system topology to distribute the data. The programmer does not need to reason in terms of the number of physical 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) Generate a virtual topology structure; (2) mapping the data grids to the different processors using chosen load-balancing techniques; (3) automatically determine inactive processors at any stage of the computation; (4) identify the neighbor processors to use them in communications; and (5) build communication patterns to be reused across algorithm iterations.

1.1.4 Towards a unified programming model

Supporting all the abstractions in a unified programming model requires a runtime system with a wider scope than the one offered by the original Hitmap library. When this research work started, the Hitmap library had only functionalities to support static hierarchical mapping over dense arrays. We needed to replace or to extend the runtime library to support other classes of parallel applications or other programming paradigms. It is at that point where the objective of this Thesis arose. This Thesis analyzes two of the most important limitations of current runtime systems, such as Hitmap:

- *Unified support for dense and sparse data.*

  Not many parallel programming frameworks transparently integrate support for both dense and sparse data structures. Most parallel programing languages only have a native support for dense arrays, including primitives and tools to deal with data locality and/or distribution only for dense data structures. Coding sparse-oriented applications with these languages implies either to manually manage the sparse data with an expensive programming effort, or to use domain specific libraries that do not follow the same conceptual approach. In both cases, the reusability of code developed previously for dense data structures is rather poor.

- *Integration of dynamic parallel paradigms and models.*

  Simple static parallel structures are easy to program with current common parallel programming solutions. However, using these solutions for programming dynamic and dataflow applications is challenging, especially in hybrid distributed- and shared-memory platforms. They offer an abstraction level too low to deal with more complex synchronization issues. Dealing with hierarchical combinations of parallel paradigms leads to highly complex codes, with many decisions about mapping hard-wired into the code.

Previous work carried out in the group had already addressed the support of heterogeneous devices in the Hitmap library [124]. The Ph.D. dissertation *Hierarchical Transparent Programming for Heterogeneous Computing* [123] presents several policies to select good execution parameters to exploit the architecture of GPU devices efficiently. It includes a study of new partitioning levels and their associated policies, data transmission across GPUs devices, task subdivision to fit into the device capabilities, and selection of appropriate shape and size for thread sets.

1.2 Objectives of this dissertation

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

> *Is it possible to create a runtime system for a generic high level programming language that offers (1) common abstractions for dense and sparse data management, and (2) generic data-mapping and data-flow parallelism support for hybrid shared- and distributed-memory environments?*

In order to be able to answer this research question, we have carried out the following specific tasks:

1. We have proposed a classification of parallel frameworks and tools into data-mapping and data-flow, depending on its execution model.
2. We have carried out an analysis of different data structures, algorithms, library tools, and programming frameworks to manage sparse data. For an application, the use of sparse-matrix and graph formats has an important impact on the performance, also requiring particular access and iterator methods.
3. We have extended the parallel programming methodology of the Hitmap library with an additional stage to select the data structure. The new methodology conceptually integrates dense and sparse domains.
4. We have redesigned the Hitmap library following the proposed methodology, adding several graph and sparse matrix representations. The new library implementation integrates dense and sparse support with the same abstractions. Different benchmarks have been implemented with this version and experimental work has been conducted to validate this solution in terms of performance and ease of programming.

5. We have carried out a study on formal modeling languages for the description of parallel systems. These kind of languages represent the dependencies of parallel programs in a formal way and have well-defined execution semantics and mathematical theories suitable for process analysis.

6. We have developed a theoretical model based on dataflow mechanisms, using a previously selected formal modeling language as its foundation. The use of an established model adds interesting properties such as analyzability. Thus, many properties of the systems, such as deadlocks, can be automatically detected.

7. We have created and implemented a framework prototype for the model introduced above. It proposes a generic form of describing a program as a reconfigurable network of activities and typed data containers arbitrarily interconnected, it extends the capabilities of previous proposed frameworks.

1.3 Research methodology

Computer Science is an interdisciplinary science, transversal to very different domains [46]. This is the reason why Computing Science researchers use several methodologies to tackle questions within the discipline [8]. The research methodology followed in the Thesis is based on a software engineering method that have four different stages: Observe existing solution, propose better solutions, build or develop, and measure and analyze [2]. 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 related literature and practical tools should be analyzed to determine possible improvements. We analyze different models and frameworks for parallel computing. After this analysis, we focus on the study of the management of sparse data and the dynamic dataflow models.

2. *Propose better solutions.*

   This phase is dedicated to the design and analysis of better solutions, to try to overcome the limits of previous proposals. We propose a new computing methodology using a unified interface for both dense and sparse data. Moreover, we create abstractions for a generic dataflow execution model based on a formal modeling language.

3. *Build or develop the solution.*

   The research methodology of this phase consists on building a prototype to demonstrate the feasibility and efficiency of the solution. The construction of the prototype must include new features identified in the previous phase and that have not been demonstrated before in other proposals. We develop our proposals as either an extension or a new version of the Hitmap library.

4. *Measure and analyze the new solution.*

   Finally, experimental work is used to evaluate the new solution.

1.4 Document structure

This document is organized as follows. Chapter 2 discusses the state of the art. It describes the most successful parallel frameworks and solutions, as well as PGAS languages, skeletons, and dataflow models. That chapter also reviews different formats and tools for sparse data support. Chapter 3 describes the Hitmap runtime library, that will be the starting point to implement a unified runtime system. In Chapter 4, we present a solution to integrate dense and sparse data management in parallel programs using a common abstraction and interface. Chapter 5 introduces a new parallel programming model and framework, based on the dataflow paradigm, that transparently supports hybrid shared- and distributed-memory systems. Finally, Chapter 6 contains the conclusions of the Thesis, enumerating its contributions and the resulting publications.

Chapter 2

State of the art

The purpose of this chapter is to introduce different parallel programming frameworks and tools that are relevant to the research work presented in this dissertation. As it was discussed in the previous chapter, the goal of the Thesis is to find out how to build a runtime system for a unified parallel framework that supports different parallel paradigms. Thus, this general system will take some parts from previous proposed solutions.

In this chapter, we make a literature review of different parallel frameworks and tools, from the most general to some particular ones. Existing frameworks are usually classified by the programming paradigm they support: Data-parallel, or task-parallel. For our goal, this is not a very clear classification because frameworks can support both types. Instead, we focus on the execution model of the frameworks. We have divided the reviewed frameworks in two categories: Frameworks focused on data-mapping and frameworks focused on data-flow. In a data-mapping execution model, all the computations are driven by the chosen mapping of data, whereas in data-flow, the execution is controlled by the data dependencies.

Moreover, in this chapter, we also discuss about Algorithmic Skeletons. This solution encapsulates usual parallel programming patterns, thus covering many common use cases.

Many of the studied proposals lack sparse support. However, these type of data structures condition many decisions about data distribution and communication. For this reason, we dedicate another section to study the problem of sparse data management in parallel programs. We discuss sparse data support solutions, including storage formats, partition libraries, and some frameworks with built-in sparse data support.

2.1 Basic parallel programming tools

There are some parallel programming tools that do not fall in any of the two categories we have proposed (data-mapping or data-flow) because they do not really offer any execution model. Instead, they are basic tools that compose the building blocks used to construct parallel systems.

These tools are classified according to the view provided for the memory address space, namely shared- or distributed-memory. For shared-memory, there are thread-based libraries, such as Pthreads [15, 62]. For distributed-memory, the basic tools are message passing models, such as MPI [66, 93].

2.1.1 Pthreads

The Pthreads API, also known as *Portable Operating System Interface (POSIX) Threads*, is a standardized programming interface for the creation and management of threads [15, 62]. The Pthreads API was specified in the IEEE standard 1003.1c-1995. It has become the standard thread management interface, since it is supported by most vendors. Pthreads is defined as a set of language programming types and procedure calls, implemented within a library.

Working with this API requires the programmer to explicitly create the threads by calling a function. Pthreads provides several synchronization mechanisms such as mutexes (mutual exclusion), condition variables, and signals. However, it is the programmer's responsibility to ensure thread synchronization and to protect the shared data variables to avoid undesired stochastic behaviors, race conditions, and deadlocks, in order to ensure program correctness.

2.1.2 Message-Passing Interface

Message-Passing Interface (MPI) is a message-passing library interface specification [66, 93]. MPI defines the syntax and semantics of library routines for portable message-passing programs. The standard defines the specification of language bindings for C and Fortran, while different vendors provide their own implementation of MPI. A MPI program consists in a set of processes that have only local memory and that communicate with other processes by sending and receiving messages. The data is moved from the local memory of one process to the memory of another process using explicit send (resp. receive) function calls in both processes. MPI also provides collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI has become the *de facto* standard for communication on distributed memory systems. Nonetheless, the use of MPI is not restricted to parallel machines with a physically-distributed memory. It can also be used for parallel architectures with a physically-shared address space like multicore architectures [106].

Using MPI has many advantages. It can be used to solve a wider range of problems than thread libraries; the codes run on either shared- or distributed-memory architectures, and the MPI vendor libraries can take advantage of architecture-dependent optimizations. On the other hand, creating a parallel application with MPI can be complex, because the programmer has to determine the data dependencies, divide the algorithm into tasks, and then implement the message passing and synchronization.

2.2 Data-mapping based approaches

The proposals discussed in this section have a data-mapping execution model. The data partition and mapping guide how the application works. The tasks to be performed are associated with memory portions and the communication and synchronization depend on that mapping.

2.2.1 High Performance Fortran

High Performance Fortran (HPF) is a high-level parallel computing language based on Fortran [71]. It was developed by the *High Performance Fortran Forum* (HPFF), a group composed by parallel machine vendors, government laboratories, and university research groups. The first version of the language specification was published in 1993.

The goal of HPF was to address the problems of writing data parallel programs for architectures where the distribution of data impacts performance. It focuses on data parallel programming, using a single thread of control and a global memory address space. The proposed language is based on Fortran 90, with extensions defined as a set of directives in the form of comments, which are ignored by uniprocessor compilers. This solution was later used in OpenMP. The directives are used to describe the collocation of data and the partitioning among processors. In addition to these features, other directives are used to express parallel computation explicitly. A *forall* directive specifies that the iterations should be executed in parallel. In HPF, all the communications are implicitly generated by the compiler, so the programmer does not need to be concerned with the details of message passing.

By the late 1990s, HPF faced many problems [82] and the community lost interest. Nevertheless, HPF introduced important ideas that have been part of newer, high-performance computing languages.

2.2.2 OpenMP 2.0

OpenMP (*Open Multi-Processing*) is a parallel programming model for shared memory multiprocessors [28, 62]. OpenMP takes a directive-based approach for supporting parallelism similar to HPF. It consists of a set of directives that describes the parallelism in the source code, along with a supporting library of subroutines. The directives are inserted into a sequential program written in a base language such as Fortran, C, or C++. They take the form of source code comments (in Fortran) or `#pragmas` (in C/C++). OpenMP simplifies the multithreading programming compared with thread libraries, such as Pthreads. OpenMP directives provide support for expressing parallelism, synchronization between threads, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization. Version 3 of OpenMP added tasking support, a feature discussed in Sect. 2.3.2.

The idea behind OpenMP is that the sequential program does not change when it is annotated with directives. Thus, programs are still valid even when they are processed by compilers that do not support these directives. OpenMP is very useful to perform data decomposition on loops. However, it is less suitable when more intricate thread synchronization is needed. Dependencies existing in the parallel code usually require synchronization. However, OpenMP does not analyze them: It is the programmer task to use the primitives and sometimes to change the sequential code to ensure that the parallel program follows sequential semantics.

2.2.3 Partitioned Global Address Space

The *Partitioned Global Address Space* (PGAS) memory model offers the programmers a virtual global shared address space. The programmer code can access variables regardless of whether they are stored in a local or remote memory. If the variable is stored remotely, the compiler and runtime are responsible for implementing the required communications. The model simplifies parallel programming while exposing data/thread locality to enhance performance. Many languages follow this model, such as UPC, Coarray Fortran, Titanium, the HPCS languages, or HTA.

The *High Productivity Computing Systems* (HPCS) is a 2002 DARPA project for developing a new generation of high productivity computing systems [41, 106]. As part of the project, three vendors – Cray, IBM, and Sun (now Oracle) – were commissioned to develop new parallel languages that would optimize software development and performance. The languages, which are based on PGAS models, are Chapel, X10, and Fortress.

We will now describe the PGAS languages in detail.

Unified Parallel C

Unified Parallel C (UPC) [23, 125] is a parallel extension to the standard C language [83]. UPC follows the PGAS programming model. The first version was published in May 1999. Version 1.3 of UPC was released in November 2013. In UPC, parallel execution is obtained by creating a number of threads at the program start. UPC uses the keyword `shared` to distinguish between data that is strictly private to a given thread and data that is shared among all threads in the parallel program. UPC declarations give programmers control over the distribution of data across threads. A thread and its associated data are typically mapped by the system into the same physical node to exploit locality.

There is no implicit synchronization among the threads running in the system. The programmer has to specify the memory consistency model for accessing shared variables for a particular statement or sequence of statements. Each data variable in the program may be annotated to be either *strict* or *relaxed* depending whether it is necessary to enforce sequential ordering of operations. In addition, UPC provides two mechanisms to realize synchronization operations at particular points: Barriers and parallel loops.

Coarray Fortran

Coarray Fortran [78, 92] is an extension to the Fortran language, which allows parallel programs to be written using shared data entities known as coarrays. It was originally proposed in 1998 as an extension to Fortran 95, and it is now part of the Fortran 2008 standard. Parallel programming with coarrays implies to execute several instances of a Fortran program, called images. Images access remote data using coarrays. A coarray is a data element, which can be a scalar or an array. Each image has a local copy of the coarray and its value can be addressable from other images using the so-called codimension. The codimension extends the bracketed tuple syntax of a regular array to access the value of the coarray on other images. The images are executed asynchronously, although some situations cause implicit synchronization of the images. Moreover, Coarray Fortran provides statements to perform explicit synchronization.

Titanium

Titanium is an explicitly parallel dialect of Java designed for high-performance scientific programming [21, 137]. This project, developed by a group from Berkeley University of California, started in 1995. Parallelism is achieved through *Single Program Multiple Data* (SPMD) and PGAS models. Instead of Java threads, Titanium uses a static thread model (Titanium threads are called processes) with partitioned address space. Each process has an area of memory that may be accessed by other processes through references, thus making an explicit distinction between local and global accesses. Titanium processes communicate through shared variables: The programmer is responsible for inserting barriers to synchronize the processes.

The more frequent point-to-point communications are done implicitly using the shared address space. Moreover, Titanium has operations for explicit communications, including two primitives for creating shared state, broadcast and exchange, and a library of operations for performing scans and reductions.

Chapel

Chapel is an emerging parallel language whose design and development have been led by Cray Inc [26]. One of the Chapel motivations is to support general parallel programming with a single and unified language. To this end, it was designed with concepts for data parallelism, task parallelism, and synchronization-based concurrent programming, which can be arbitrarily composed.

Chapel has a dynamic multi-threaded execution model that is more general than a *Single Program Multiple Data* (SPMD) model. Users express parallelism within their programs in terms of tasks that should be executed in parallel. The tasks are executed by the threads, potentially creating additional tasks. The synchronization of computations on shared data is done through special constructions or using shared variables.

Another Chapel concept is the use of a global view for data structures and control flow. The programmer can declare and compute on distributed data structures in the same way as for a completely local version. Chapel allow programmers to specify how arrays are distributed among physical nodes. For this Chapel defines several languages elements, such as: *domain*, *locale*, and *domain map*. A Chapel domain represents an array index set; a locale represents a computational unit of the target system architecture; and a domain map specifies how to map the domain's indices and array's elements in the locales. A domain map that only references one locale is called *layout*, while a domain map that target multiple locales is called *distribution*. Chapel has a plug-in system, aimed for advanced users, that can be used to create new domain maps to control the distribution and layout of domains and arrays.

X10

X10 [110, 111] is a parallel programming language developed by IBM as an extension to Java, designed specifically for high performance computing. X10 follows the *Asynchronous Partitioned Global Address Space* (APGAS) model [112]. This model extends the standard PGAS model by introducing logical places. An X10 place corresponds to a single operating system process, each with one or more simultaneous threads (activities). Places permit the programmer to explicitly deal with notions of locality. There are two levels of concurrency. The threads of a place have a locally synchronous view of their shared address space, while threads of different places work asynchronously with each other. X10 activities can only directly access memory from the place where they live; they should use constructs to access memory at remote places. X10 places are similar to Chapel's locales. However, in Chapel the access to remote memory spaces is transparent. The Chapel tasks executed within a given locale can access variables regardless the locale where they were declared.

Fortress

Fortress [7] is an object-oriented language inspired by Fortran that have been developed by Oracle (formerly Sun Microsystems). It was part of the DARPA HPCS project until it was discontinued in 2006. This led to uncertainty about its future. Finally, in 2012, the Oracle Labs Programming Language Research Group winded down the Fortress project.

Fortress mimics mathematical notation and provides syntactic support for summations, productions, and other operations. Fortress supports the parallel execution of programs with many implicitly parallel constructs. For example, loop iterations, method invocations, and the evaluation of function arguments are parallel by default. Moreover, all arrays are distributed. Additionally, explicit threads can be created for the execution of program parts. In order to control interactions of parallel executions, Fortress includes atomic expressions. An atomic expression completes in a single step relative to other threads. Every object in a Fortress program is considered to be either shared or local. A local object may be accessed faster than a shared object, particularly in the case of atomic reads and writes.

Hierarchically tiled arrays

Hierarchically tiled arrays (HTAs) [44, 45] is an object-oriented parallel programming library that provides a class to create and manage arrays partitioned into tiles. The tiles can be either conventional arrays or they can be recursively tiled. Tiles can be distributed across processes in a distributed-memory machine. All the processes know the structure of the HTAs. However, they only store the data of the tiles in the distributed HTAs that they own. Synchronization will take place on demand when data from other process is needed.

HTAs encapsulate the parallelism providing a global view of the distributed data. They allow the parallel computation and data movement to be expressed by means of indexed assignment and computation operators. HTA operations include standard arithmetic operations, permutations, reductions, and partitions to modify the HTA tiling structure. There are implementations in Matlab and C++.

2.3 Data-flow based approaches

The second group in our literature review is dedicated to approaches that are based on data-flow execution models. In this case, the computation is driven by data dependencies. The load balancing is done by a scheduler instead of being associated to the data partition. The data partition only affects the task granularity.

The frameworks discussed in this section support the stream programming paradigm. In this paradigm, several parallel stages are connected through data channels used to communicate tasks.

2.3.1 FastFlow

FastFlow [5] is a structured parallel programming framework targeting shared memory multi-core architectures. FastFlow is structured as a stack of layers that provide different levels of abstraction, providing the parallel programmer with a set of ready-to-use, parametric algorithmic skeletons, modeling the most common parallelism exploitation patterns (see Fig. 2.1). The lowest layer of FastFlow provides efficient lock-free synchronization mechanisms to implement efficient *Single Producer, Single Consumer* (SPSC) FIFO (*First In First Out*) queues. The middle layer extends the previous one to *Multiple Processor, Multiple*

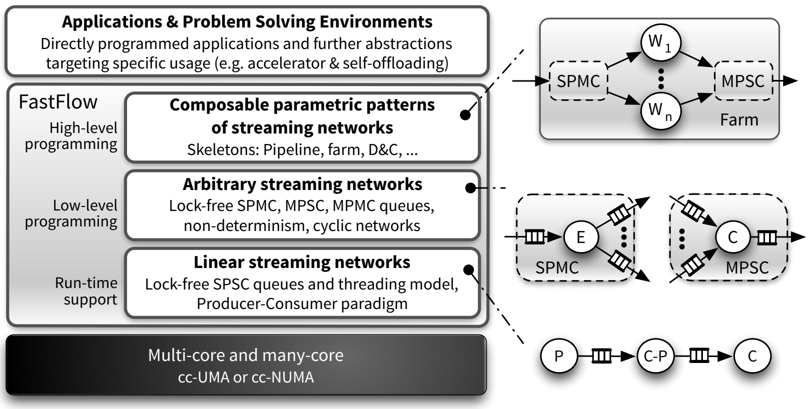

Applications & Problem Solving Environments
Directly programmed applications and further abstractions targeting specific usage (e.g. accelerator & self-offloading)
FastFlow
High-level programming
Composable parametric patterns of streaming networks
Skeletons: Pipeline, farm, D&C, ...
Low-level programming
Arbitrary streaming networks
Lock-free SPMC, MPSC, MPMC queues, non-determinism, cyclic networks
Run-time support
Linear streaming networks
Lock-free SPSC queues and threading model, Producer-Consumer paradigm
Multi-core and many-core
cc-UMA or cc-NUMA
SPMC
W1
Wn
MPSC
Farm
E
SPMC
C
MPSC
P
C-P
C

Figure 2.1: FastFlow layered architecture diagram with abstraction examples at different layers of the stack [5].

Consumer (MPMC) queues, which are implemented using only SPSC queues and control threads, thus providing lock-free and wait-free arbitrary data-flow networks with low synchronization overhead. Finally, the top layer provides, as programming primitives, typical streaming patterns exploiting the fast communication provided by the lower layers. Rather than allowing programmers to connect stages into arbitrary networks, FastFlow provides high-level constructs such as pipeline, farm, and loop.

The FastFlow group is working on a distributed multi-core extension using a two-tier model [4]. Inside a node, the extension uses a lower tier with a shared-memory implementation of skeletons. This is combined with an upper tier of structured coordination among the set of distributed nodes executing the lower tier computations. This solution forces the programmer to manually divide the program structure for the available memory spaces, and to use a different mechanisms of external data channels to communicate the tasks.

2.3.2 OpenMP 3.0

Version 2.0 of OpenMP was primarily suited to loop-level parallelism. Version 3.0 [97], published in 2007, added the concept of tasks into the OpenMP execution model. Although not explicitly, the previous version did have task support [91], because loop iterations were divided into tasks that were then feed to the available threads. The main contributions of version 3.0 was the ability to create explicit tasks using a new directive. While this allows more general programming approaches, the limitation of this solution is that the tasks are independent. Thus, programmers have to still take care of the synchronization between threads using different mechanisms such as barriers, critical sections, or atomic variables.

2.3.3 OpenStream

OpenStream [103] is a dataflow OpenMP extension where dynamic independent tasks communicate through streams. The programmer exposes data flow information using directives to define the input and output task of the stream. This allows arbitrary dependence patterns between tasks to be created. A stream can have several producers and consumers that access the data in the stream using a sliding window. The OpenStream runtime ensures the coordination of the different elements. Compared with standard OpenMP, in OpenStream the programmers do need to explicitly ensure the synchronization between threads.

2.3.4 S-Net

S-Net [64, 65] is a declarative coordination language. It defines the structure of a program as a set of connected asynchronous components called boxes. S-Net only takes care of the coordination: The operations done inside boxes are defined using conventional languages. Boxes are stateless components with only a single input and a single output stream. The idea behind S-Net is to separate the design of the sequential parts of the application from the coordination of the parallel algorithm. Thus, the parallel programmer can focus on the concurrency problems, taking into account the target architectures, and using the boxes as building blocks for the parallel application.

The original S-Net implementation was designed for shared-memory architectures. There is also a distributed extension [63] that allows programmers to work with cluster of multi-core machines. Following the S-Net methodology, the programmer has the responsibility to map portions of the network onto the distributed nodes. Inside a node, the network components interact via shared-memory, whereas message passing is used between components on different nodes. From the programmers' perspective, the implementation of streams on the language level by either shared memory buffers or distributed memory message passing is entirely transparent.

2.4 Algorithmic Skeletons

Algorithmic skeletons are a high-level parallel programming approach introduced by Cole [33, 34] that takes advantage of common programming patterns to hide the complexity of parallel programs. The algorithmic skeletons could be seen as higher-order functions that act as templates for the computation structure of a program. The parameters of these functions are other functions that specify the actual computation of the problem. Using this method, programmers do not have to write the code to perform the communications or synchronizations between processes. They only have to provide the specific code to solve the problem, using the template provided by the skeleton.

Since skeletons provide solutions for well-known classes of parallel problems, a unified programming system that supports different parallel paradigms would likely support these classes of problems. Thus, the underlining runtime for that system would take some of the solutions that are found in skeleton frameworks.

The use of parallel skeletons has several advantages:

- Skeletons simplify the programming task by using a higher level of abstraction. The programmer has no longer to manage explicitly the communication of data or tasks in the program.
- This model is less error-prone than traditional parallel models because the coordination and communication is controlled by the skeleton implementations.
- Skeletons do not impose a performance penalty. A efficient skeleton implementation can obtain the same performance as one based on a lower-level model.
- Skeletons are portable, because the details of communication for the patterns are hidden by the implementation. It is also possible to provide a well tuned implementation for a particular architecture.
- They can also drive the utilization of optimizations because they make it possible to extract information about data dependencies that could not be represented using other models.
- Skeleton applications can be also optimized by using cost models to schedule tasks and resources.

Among the limitations of skeletons, one of the most important is that the frameworks that implement them only support a limited number of patterns. Thus, if the provided patterns do not match the algorithm of the developed application, they simply cannot be used. The same happens when changes in the skeleton behavior is required, Although, they might be seen as small conceptual changes, if they are hardcoded in the implementation they cannot be done.

2.4.1 Skeleton classifying

Skeletons are generally classified as *data parallel* or *task parallel*. Previous surveys add an extra category named *resolution skeletons* [31, 60]. Data parallel skeletons are viewed as higher-order functions applied to data structures. Internally, they manipulate the elements according with computation patterns in a finer grain. Task parallel skeletons compute workflows of tasks where the behavior is determined by the interaction between tasks. Resolution skeletons solve a family of problems with iterative phases of computation, communication, and control.

	Static	Dynamic
Data parallel	map, fork, zip, reduce, scan, stencil	-
Task parallel	sequential, pipeline, wavefront	farm
Resolution	-	divide and conquer (D&C), branch and bound (B&B), mapreduce

Table 2.1: Algorithm skeletons taxonomy.

We propose to add another dimension to this skeleton classification depending on the nature of their computation structure. We add two categories: *Static* and *dynamic*. Static skeletons maintain the same structure during all their execution, whereas dynamic skeletons have a mutable computation structure. Static skeletons can take advantage of static scheduling methods, pre-calculated during the initialization phase, while dynamic skeletons need dynamic scheduling and load balancing techniques.

Table 2.1 shows the relationship between both classifications. Data parallel skeletons are static, while resolutions skeletons are dynamic; because their computation structure depend on the particular data being processed. Finally, we find in the task-parallel skeletons class both static and dynamic examples.

There is a direct relationship between our skeleton dimension and the execution model of the frameworks that implement them. Static skeletons are solved by frameworks with a data-mapping approach, while dynamic skeletons require data-flow frameworks.

In the following paragraphs, we review and classify the most usual parallel skeletons that appears in the literature. It is not intended as a comprehensive study: More information about parallel skeletons can be found in [60] or [61].

Data parallel skeletons

In a data parallel skeleton, the same operation is performed in parallel on different elements in a data structure.

Map The *map* skeleton is the simplest one. This skeleton takes a unary function (or sub-skeleton) and applies it to all elements of a given data structure. The result is another structure with the same shape as the input. For each input element, it places the result in the correspondent position of the result structure. The function can be applied simultaneously to every element to achieve parallelism because computations are independent from each other.

Fork The *fork* skeleton is similar to *map*, but, instead of applying the same function to all the elements of the data structure, a different one is applied to each element.

Zip The *zip* skeleton combines two structurally-identical data structures by applying a binary operator to each corresponding element. The result of this skeleton is another structure with the same shape. It can be viewed as an extension of the *map* skeleton.

Reduce The *reduce* skeleton (aka *fold*) combines the elements of a data structure using an associative and bijective operation, and returns the combined value. It traverses the data structure and applies the operator to each element. The typical operations are addition, maximum, or minimum.

Scan The *scan* skeleton computes the partial reduction of each element by reducing all previous elements. Note that, as opposed to *reduce*, it maintains aggregated partial results. Prefix sums are its most representative example.

Stencil The *stencil* [113] is a common kernel in many scientific applications. It performs nearest neighbor computations over a spatial grid. This pattern appears for example in the cellular automaton, a computational system defined as a collection of cells that change state in parallel based on a transition rule applied to each cell and its neighbors [114, 117]. A stencil skeleton executes an update function over the data grid taking into account the data communications. For example, the eSkel library [34] provides a haloswap skeleton that can be used to implement a stencil.

Task Parallelism

Data parallelism refers to the distribution of data. Instead, task parallelism refers to the distribution of computing. In task parallel skeletons, each computing unit performs an operation and communicates data with each other inside a workflow.

Sequential A *sequential* skeleton defines an atomic operation that can be executed sequentially. They are the final leaves used in skeleton nesting. This skeleton could be further classified depending whether it produces new tasks, consumes them, or both. For example, Muesli [31] has *initial*, *final*, and *atomic* sequential task parallel skeletons.

Pipe A *pipe* skeleton enables staged (pipelined) computations. It is composed of a set of stages connected in series. The output of one stage is the input of the next one. The parallelism is achieved by computing the different stages simultaneously on different tasks. The time of processing a single task is normally increased by the pipeline. However, the overall performance is improved by overlapping the computation across the pipeline.

Wavefront A *wavefront* skeleton is a multidimensional variant of a traditional pipeline. Sometimes, this skeleton is included inside the category of data parallel, but there are also task-based implementations of the wavefront pattern [75]. In this pattern, data elements are distributed on a multidimensional grid. The elements must be computed in order because there are dependencies between elements in one direction for each dimension. The computation resembles a diagonal sweep across the elements of the grid. Starting from a singular corner of the grid, each element performs its calculation and sends data to the following elements, thus propagating their effect diagonally.

Farm A *farm* skeleton, also known as master-slave/worker, consists of a farmer and several workers. The farmer receives a sequence of independent tasks and schedules them across the workers. The workers execute the task and send the results back to the farmer.

There are several implementation schemes for the farm skeleton [101]. The straightforward implementation consists of a single farmer process and several workers. The farmer controls all the scheduling logic, sending tasks to idle workers. This scheme has the disadvantage that the farmer might become a bottleneck due to the great amount of messages exchanged between farmer and workers.

Different task scheduling could be applied. For example, random, cyclic, or load balancing. A farm where the tasks are distributed cyclically to workers is also referred as a *deal* skeleton [18]. A *deal* skeleton is appropriate when there is a big number of task and the workload is homogeneous.

Parallel The *parallel* skeleton is similar to the *farm*. However, each input is forwarded to all the workers [101].

Resolution skeletons

Resolution skeletons are designed to solve a particular family of problems. They are more complex than the parallel task. The user do not only have to provide the function to solve a single task, but also to provide methods to generate, distribute, combine, and/or evaluate tasks depending on the problem.

Divide & Conquer *Divide and Conquer* (D&C) is a classical skeleton, in which the solution to a problem is obtained by dividing the original problem into smaller sub-problems and then the same method is again applied recursively [102]. The recursive division is controlled by a condition that determines whether the current sub-problem must be divided again or it is small enough to be solved. Simple problems are solved directly. Then, the solutions obtained at each level are combined using a composition operator to obtain the final solution for the whole problem.

The user provides four basic operators: *test*, *divide*, *combine*, and *solve*. The test operator checks if the given problem can be solved directly or it needs to be divided. Depending on the result of the test operator, either the problem is divided in several sub-problems using the divide operator, or it is passed to the solve operator to get a result. Finally, the combine operator constructs the final solution by combining the partial solutions.

Branch & Bound *Branch & Bound* (B&B) is a general algorithm to solve certain optimization problems [100]. It explores the complete search space, looking for the optimal solution. The algorithm can not enumerate all possible solutions, because their number usually grows exponentially. Instead, it divides the search space recursively (*branch*) generating unexplored subsets. Then, it discharges the subsets that will not lead to the solution using an objective function (*bound*).

A B&B implementation has a work pool with unexplored subsets. It initially contains only a element specifying the complete problem. In each iteration of the algorithm, a subset is chosen from the pool and divided into smaller subsets using a branching function. Then the bound function is applied to the subsets. Only subsets with admissible solutions are included into the work pool for the following iteration.

MapReduce *MapReduce* is a skeleton used for processing large data sets. Its data processing model was first introduced by Google's MapReduce framework [39, 85]. There are other implementations of MapReduce, such as the open-source framework Hadoop [72]. The skeleton has two phases (*map* and *reduce*) that do not have a direct correspondence with the map and reduce skeletons. During the map phase, the input data is processed to generate key/value pairs. The reduce phase produces the final output by merging together all the intermediate pairs associated with the same key. MapReduce takes advantage of the parallelism in several stages. Since key/value pairs are independent, map invocations can be distributed across multiple processes by partitioning the input into multiple sets. There is also parallelism in the reduce phase because the reduce invocations are performed independently for each set with the same key.

2.4.2 Skeleton frameworks

In this section we will study specific parallel skeleton implementations. We are interested in the execution models they implement to support the provided skeleton set. This is not intended to be a comprehensive study but a description of the state of the art. A more complete survey of skeleton frameworks can be found in [60].

Calcium and Skandium

Calcium [25] is a skeleton framework written in Java. It supports basic data- and task-parallel skeletons that can be combined and nested to solve more complex problems. Calcium is part of the ProActive Middleware [24], a grid programming platform.

Skandium [86] is a reimplementation of Calcium for multi-core architectures. The execution environment of Skandium uses a producer/consumer model. The library has a pool of workers that consume task from a ready queue and compute it following the instructions of the skeleton. When a task is finished, the result is passed to the user. A task may generate new sub-tasks that will be inserted into the ready queue. In this situation, the main task will wait until its sub-task are completed. A waiting task will be reinserted in the ready queue when all its sub-tasks are finished.

Threading Building Blocks

Threading Building Blocks (TBB) is a threading abstraction library developed by Intel to take advantage of multi-core architectures [104, 107]. TBB provides a portable implementation of parallel patterns, such as *for*, *reduce*, *scan*, *while*, *pipeline*, and *sort*. The TBB *sort* pattern is a special case of *divide & conquer*. It also offers thread-safe data containers (hash map, vector, and queue) and synchronization primitives. The core of the library is a thread pool managed by a task scheduler. The scheduler maps tasks onto threads efficiently and achieves load balancing using a work-stealing algorithm. To use the library, the programmer specifies tasks. Using TBB tasks is often simpler and more efficient than using threads, because the task scheduler takes care of the details.

eSkel

The *Edinburgh Skeleton Library* (eSkel) [18, 34] is a C library that internally uses MPI to execute communications. Thus, it inherits the MPI model of SPMD distributed memory parallelism. It defines five skeletons: *pipeline*, *farm*, *deal*, *haloSwap*, and *butterfly* (particular case of *divide & conquer*). In eSkel, each skeleton is a collective operation, the participant processes are grouped constituting an activity. The skeleton distributes the tasks dynamically among the processes, and handles all the interactions according to the semantics of the skeleton.

The eSkel library has different modes for nesting and interaction [17]. Nesting mode can either be transient or persistent, while interaction can either be implicit or explicit. In a transient nesting, each invocation of the nested skeleton is initiated by the outer skeleton and destroyed afterwards, thus it is independent of the outer level. Persistent means that the skeleton is created once and used through all the application, so it persists across outer level interactions. The concept of interaction mode concerns how the data flows between skeletons. The implicit interaction mode means that the flow of data is defined by the skeleton composition. In this mode, a skeleton output will trigger an interaction, and another skeleton will receive it as an input. On the contrary, explicit interaction mode means that interactions can be triggered without following the skeleton composition. For example, this is the case of a generator stage in a pipeline that can generate output tasks without an external trigger.

Muesli

The Münster Skeleton Library (Muesli) [31, 100, 101] offers data and task parallel skeletons through a library implemented in C++, and uses MPI to communicate data between processes. Recent versions of Muesli also support multi-core programming with OpenMP. The supported skeletons are *fold*, *map*, *scan*, and *zip* for data parallelism. These skeletons manipulate a distributed data structure as a whole, but they are internally implemented in parallel. For task parallelism, Muesli provides *pipeline*, *branch & bound* and *divide & conquer*. It inherits the two-tier model of P3L [10], where data parallel skeletons could be nested inside task parallel ones.

Muesli offers a sequential programming style and hides all the coordination details inside the library, not being necessary to deal with the MPI routines. It takes advantage of the C++ language using polymorphic types to construct the skeletons.

SkeTo

SkeTo (*Skeletons in Tokyo*) [89] is a skeleton library for distributed environments implemented in standard C++ with MPI. This library provides a set of data parallel skeletons based on the theory of Constructive Algorithmics. In Constructive Algorithmics, the program control structure is derived from the data structure it manipulates. One of its important results is the systematic program optimization by fusion transformation. This transformation merges two function calls into a single one, eliminating the overhead of both function calls and the generation of intermediate data structures. SkeTo fuses skeleton calls statically at the compilation time. A newer version of the SkeTo library [79], targeted for clusters of multi-core nodes, implements a dynamic fusion of skeleton calls at the library level using a template technique in C++ language. This makes possible to fuse a wider number of skeleton function calls.

SkelCL

SkelCL [120] is a GPU skeleton library based on data-parallel algorithmic skeletons. It provides a set of basic skeletons (*map*, *zip*, *reduce*, and *scan*). SkelCL generates OpenCL [84] code (kernel functions) from skeletons, which is then compiled by OpenCL. User-defined customizing functions passed to the skeletons are merged with pre-implemented skeleton code during code generation. It also provides some support to program multiple GPU devices.

2.4.3 Summary of skeleton functionalities

There are several design concepts that have to be taken into account when developing a skeleton framework. This section collects the details introduced previously in the literature. They are relevant for the Thesis because they will help us to define the characteristic needed by our new runtime library.

Nesting mode If a skeleton uses internally another one, there are two possible nesting modes: transient or persistent [17, 18]. In a transient nesting, the outer skeleton calls an inner one to process some internal data. The inner skeleton only exists during the invocation of the external stage. A new instance is created each time. In a persistent nesting, the input and output of the outer skeleton are mapped to the inner one. The instance is persistent between invocations.

Interaction mode This concept defines the relationship between the skeleton input and output. There are two possible interaction modes: implicit and explicit [18]. In an implicit interaction mode, a skeleton produces an output for each consumed input. In an explicit interaction mode, a stage in the skeleton can produce an output arbitrarily without a previous input. Moreover, a skeleton can process an input without producing a result.

Task scheduler Several skeletons such as *farm* or *divide&conquer* are composed of a set of workers. These kinds of skeletons need a mechanism to send the tasks to workers and to collect the results. The use of a dispatcher and a collector is one of the possible solutions. However, it has been shown that this solution does not always achieve a good performance [101]. Instead, distributed solutions, such as the TBB scheduler [107], or a distributed work pool [100], are preferred because they avoid the contention and the bottlenecks that may arise with the use of a centralized scheme.

Task distribution policies A work pool requires a distribution policy to assign tasks to workers. Since the time required to process a particular task is usually not known, many work pools assume that each one requires the same time. Under these conditions, there are two independent distribution policies: Random and cyclic. Both policies lead to similar performance when there are a big number of tasks. However, a cyclic distribution performs a fairer distribution when the number of tasks is small [101]. More complex schemes with load balancing can be applied if there is information about the actual load of each task and the capabilities of the worker.

2.5 Sparse data support

Many applications work with matrices where most of the elements are zero. These matrices are called *sparse matrices*. Meanwhile, a graph can by represented by its adjacency matrix, where each element determines if there is a connection between two given vertices. Since adjacency matrices are usually sparse, the management of sparse matrices and graphs shares the same techniques. There are special storage formats and algorithms for these kind of structures. The main feature of these storage formats is that they do not store zeros – or they only store a small number of them – so they use much less memory. Moreover, particular algorithms can exploit the sparsity of the matrix, making applications much more efficient with respect to computation time. For example, algorithms for sparse matrix multiplication reduce the asymptotical complexity compared with dense algorithms [138].

In a parallel program, the use of sparse structures requires different methods to distribute the load, and changes in the code to access the data efficiently. For this reason, we will discuss the common storage formats used for sparse matrices, the partitioning algorithms used for parallel programming, and frameworks with built-in support for sparse data structures.

2.5.1 Sparse formats

For sparse matrices, storing the zero elements would waste a lot of memory space. The sparse matrix storage formats aim to reduce the needed space by grouping the nonzero elements and discarding as many zero elements as possible. A sparse matrix format is defined by:

- The nonzero elements of the matrix, and perhaps a limited number of zeros.
- A scheme to know where the elements were located in the original matrix.

Using a scheme to localize the original positions of the matrix elements involve an increment in the cost of the access methods. Thus, a compromise between space reduction and efficient location is needed. There are specific formats that are very efficient for some particular matrix problems. These formats take advantage of a particular sparsity pattern of the sparse matrix. For example, if we know that a matrix has only elements in a few diagonals, we could store these diagonal as vectors. When the matrices do not have a regular structure, there are more general formats that can be used.

Classical data structures for sparse data [3, 115] follow a different approach. They aim to reduce the cost of operations such as vertex addition or deletion. Therefore, they are implemented using linked lists, which allow efficient modifications in the structure. The pointers needed for the linked lists increase the memory requirements. Also, since the vertices are not in contiguous memory locations, the performance when traversing all the matrix is lesser due to cache issues. Such structures are not appropriate for high performance parallel applications and so they are not covered in this work.

On the other hand, there is a great number of available sparse matrix formats with better properties to exploit parallelism. They have been widely discussed in the literature. For example, some representative ones are discussed in [16, 109, 118, 119]. We will describe the most common ones. These formats use a compact representation to minimize the memory usage and to improve the efficiency of the applications.

Note that, in the following example matrices, we use the row-wise storage format as in the C language, where the matrix uses the row as first dimension and the column as the second. Also, the first index is always 0.

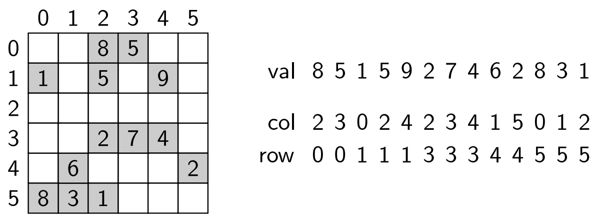

val 8 5 1 5 9 2 7 4 6 2 8 3 1
col 2 3 0 2 4 2 3 4 1 5 0 1 2
row 0 0 1 1 1 3 3 3 4 4 5 5 5

Figure 2.2: Coordinate list storage format example.

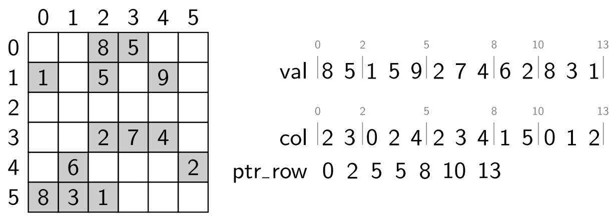

val 8 5 | 1 5 9 | 2 7 4 | 6 2 | 8 3 1
col 2 3 | 0 2 4 | 2 3 4 | 1 5 | 0 1 2
ptr_row 0 2 5 5 8 10 13

Figure 2.3: Compressed Sparse Row storage format example.

Coordinate list

COOrdinate list (COO) format is maybe the most simple storage format for sparse matrices. Figure 2.2 shows an example of this format. It consists in a list of coordinates with the nonzero values. It requires three vectors: A vector with the same datatype as the matrix elements for the nnz nonzero values, and two integer vectors with the coordinates associated to the elements. In each entry, the first vector contains the a_{ij} value, and the integer vectors the i and j index respectively.

The size of this storage format is $3nnz$. It also needs a single data element to store the nnz value. It is a very flexible format when additions of elements are needed, but it requires a longer amount of space compared with other formats. Moreover, the access of an element involves a search in the list that has a computational cost in $O(n)$ when the elements are not sorted. If the elements are sorted (e.g., first by row index, then column index), the random access time is reduced to $O(\log n)$ because a dichotomic search can be used.

Compressed Sparse Row

Compressed Sparse Row (CSR) is one of the most general formats. It makes no assumptions about the sparsity of the matrix. It does not store any zero elements, but an indirect addressing step is needed for each data element. To store a matrix A with n^2 elements where nnz elements are nonzeros, this approach uses tree different vectors. An example of the CSR format is shown in Fig. 2.3.

- val is a vector that keeps the values of all the nonzero elements together, as they appear in row-major order. There is no particular order with respect to column number. The size of this vector is nnz.
- col is an integer vector that stores the column index of each val element. The column indices are grouped by row. The size of this vector is nnz.
- The last vector, ptr_row, is an integer vector that is used to locate the begin and the end of each row in the col vector. The i row starts at the value contained in $ptr\_row(i)$ and ends at $ptr\_row(i+1)-1$. The size of this vector is $n+1$.

The total size used in this format is $2nnz+n+1$. To access one element a_{ij}, we need to get the begin column index and the end column index of the given row i using the ptr_row vector. Then we should perform a search to locate the needed column index in the col vector, since as we said before, the columns may not be ordered. Thus, the access has a linear cost in the number of column elements. But if a sorting is enforced, we could save operations in the search, for example using a dichotomic search. Finally, we can get the a_{ij} value from the val vector, using the same index.

A disadvantage of this compact storage format is that it does not provide any mutability. Edges cannot easily be added or deleted. This format is used in applications with large matrices that do not need to change during the whole application.

Compressed Sparse Column

Compressed Sparse Column (CSC) is a similar format to CSR where the columns are stored instead of the rows. It is equal to CSR when the matrix to store is A^t. This format may be more efficient when the programming language uses column-wise storage, or depending on the specific matrix structure.

Block Compressed Sparse Row

The *Block Compressed Sparse Row* (BCSR) format is a non-general format used to exploit block patterns in a sparse matrix. Block matrices are common in all areas of scientific computing related to the discretization of partial differential equations [119].

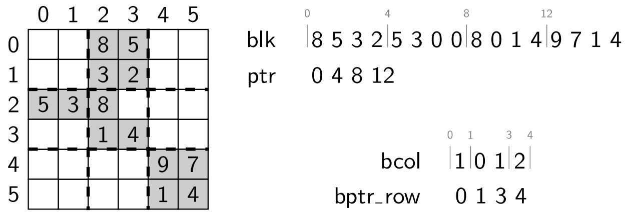


Figure 2.4: Block Compressed Sparse Row storage format example.

The full matrix is divided into blocks. This approach treats each block as a dense matrix, even when they may have zero elements, as it is shown in the example in Fig. 2.4. This format is an extension of CSR, where the elements are not simple values of the matrix, but blocks with at least one element that is nonzero. If the block dimension is *bd*, and $nnzb$ is the number of nonzero blocks in the matrix, the block dimension of A is *br* (*br* $= n/$*bd*). We need four vectors, one more that in CSR format:

- blk: A vector containing the elements of each block. The blocks can be stored row-wise, or use another chosen form. The size of this vector is $nnzb \times bd^2$.
- ptr is a integer vector that keeps a pointer to the blocks in the blk vector. It is the equivalent to the val vector in CSR, but indexing sub-matrices instead of single elements. Like in val vector, the indices are stored together, row by row. The size of this vector is $nnzb$.
- *bcol* is a integer vector that stores the column index of each ptr block. The size of this vector is $nnzb$.
- The last vector is $bprt\_row$, an integer vector that is used to locate the begin and the end of each row of blocks in *bcol*. The size of this vector is *br* $+ 1$.

This format offers a great space saving because it does not need to store all the information for each non-zero element. It can take advantage of efficient operations in dense blocks due to a more efficient cache memory usage. However, it has some drawbacks: We need to define the block size, and small dense blocks are not so efficient. Also, there exists a trade-off between the speed-up derived from an efficient cache use and the time and space lost to store zeros inside the dense blocks when the block size increases [126].

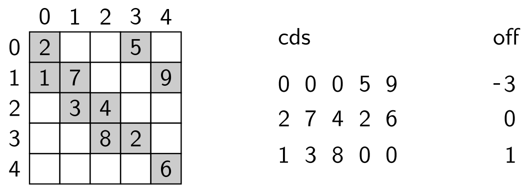


Figure 2.5: Compressed Diagonal Storage format example.

Variable Block Compressed Sparse Row

While the BCSR format uses a fixed block size, *Variable Block Compressed Sparse Row* uses different block sizes. It needs two additional vectors to store the size of the blocks. The algorithm to retrieve a particular element is more complex than in previous formats. However, with different block sizes, it is possible to represent better the nonzero pattern of the matrix, minimizing the zero elements in the stored blocks.

Compressed Diagonal Storage

The *Compressed Diagonal Storage* (CDS) format, also named *DIAgonal format* (DIA), is a non-general format used for banded matrices. A matrix is said to be banded when all its nonzero elements are confined within a band formed by diagonals parallel to the main diagonal [118].

The data structure for a compressed diagonal storage consists of a matrix and an offset vector. For a sparse matrix with n rows and m columns, the matrix $cds(ndiag, m)$ stores the $ndiag$ nonzero diagonals in consecutive locations. The vector off stores the offset of each diagonal with respect to the main diagonal. It is used to retrieve the elements. The position (i, k) of the matrix cds contains the element $a_{i,i+off(k)}$. Although the access requires several indirections, no searches are involved, thus the access cost is constant. Figure 2.5 shows an example of the CDS format.

2.5.2 Sparse matrix file formats

All the previous formats are data structures that aim to organize the data in memory so it can be accessed efficiently. Now, we will discuss some sparse matrix file formats. These formats are designed to facilitate the exchange of sparse matrix data.

Harwell-Boeing

This is a particular format used in the Harwell-Boeing sparse matrix collection [42]. They defined two compact formats in the collection. We are going to discuss the standard sparse matrix format. The second format is used for unassembled finite-element matrices. The standard matrix is stored in an ASCII file using the Compressed Sparse Column format. It has a header block that contains summary information about the storage format and space requirements, such as matrix type, matrix size, and number of lines of the file. Since it is Fortran-oriented, the format is fixed-length and the header defines the variable formats for the data block.

Matrix Market

This is a simple format used by the Matrix Marker sparse matrix database [20]. Their objective is to define a minimal ASCII file format which can be very easily explained and parsed. They have two specifications, an array format for representing general dense matrices, and a sparse format. The Matrix Market sparse format uses a Coordinate list format. It has a header to represent the matrix. Then, it stores a sparse matrix as a set of triplets with the row index, column index, and the associated value.

2.5.3 Sparse structures partitioning

Algorithms that find good partitions of graphs and sparse matrices are critical for developing efficient solutions for a wide range of problems in many application areas. Usually, these problems require the distribution of the sparse data to the processes. Examples include simulations based on finite element methods, methods to solve sparse systems of linear equations, or several optimization problems [80]. Partitioning methods designed for graphs can be also used to partition sparse matrices.

Let P be an integer number representing a number of parts. Graph partitioning consists in assign each vertex of a graph G to one of the P parts. In general, desired properties of the partition include that the number of vertices on each part is balanced, and that the number of edges connecting vertices in different partitions is minimized.

There exist two different commonly used approaches [94]: Direct methods that partition the graph directly into the desired amount of partitions, or recursive methods that split the graph into two partitions that are then divided recursively. Given a bisection algorithm to split a graph in two partitions, an algorithm to obtain several partitions can be easily created. These methods are simpler than the direct methods. However, they do not lead to optimal solutions.

Multilevel partitioning

The optimal graph partitioning problem is NP-complete [43]. Thus, it is preferred to find an approximate solution in an acceptable execution time. One of the most used approaches is the multilevel partitioning [69]. A multilevel algorithm provides reasonable good partitioning, with moderate computational complexity. These are the reason because these methods have displaced the other solutions in all the available partition libraries.

Multilevel partitioning algorithms have three phases. First, they reduce the graph size, constructing a coarse approximation of the graph by collapsing vertices and edges. This process is repeated creating a sequence of graph levels. Second, they solve partition problem for the smallest graph in the sequence. For this problem any partitioning algorithm could be used. Third, they project the coarse partition back through the sequence of graphs to construct a partition for the original graph. Vertices in thinner levels are assigned to partitions according to their representatives in coarser levels. In this phase, a refinement is applied to further enhance the current solution.

2.5.4 Graph partitioning libraries

This section describes some of the well-known available software packages for graph partitioning:

- Metis is a serial software package for partitioning large irregular graphs, partitioning large meshes, and computing fill-reducing orderings of sparse matrices [80]. It uses a multilevel k-way partitioning scheme. ParMetis [81] is a parallel version of Metis that obtains high-quality partitionings on distributed graphs. It also uses multilevel partitioning scheme with an additional method to eliminate unnecessary vertex movement during the refinement. Metis and ParMetis are developed at the Department of Computer Science & Engineering at the University of Minnesota.

- Jostle is a parallel multilevel graph-partitioning software package written at the University of Greenwich [129, 130]. It is designed to partition unstructured meshes on distributed-memory parallel computers. It can also be used to repartition and load-balance existing partitions. Jostle is commercialized under the name of NetWorks.

- Scotch is a project carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux I. It is now included in the ScALApplix project of INRIA Bordeaux Sud-Ouest. Its goal is to study the applications of graph theory to scientific computing. It has resulted in the development of the Dual Recursive Bipartitioning mapping algorithm and in the study of several graph bipartitioning heuristics, all of which have been implemented in the Scotch software package [98].

- Party is a partitioning library that provides efficiency methods for graph partitioning [94]. It provides a great variety of global and local methods with different heuristics. It is developed at the University of Paderborn.

- Chaco is a software package that provides several innovative algorithms for decomposition problems. It is developed at Sandia National Labs. Chaco allows for recursive application of several methods for finding small edge separators in weighted graphs [70].

2.5.5 Frameworks with sparse support

The management of sparse data structures is usually programmed manually. There are some frameworks that support sparse vectors and matrices. However, they are not general purpose frameworks, but rather specialized libraries for linear algebra, with kernels for iterative solvers or matrix factorizations. These libraries use the most suitable storage formats for the algorithms they implement. Since they are focused on scientific and engineering problems, they usually offer parallel solutions. A survey of these frameworks can be found in [40].

PETSc

PETSc (*Portable Extensible Toolkit for Scientific computation*) [13, 14] is a library for the parallel solution of linear and nonlinear systems of equations arising from the discretization of partial differential equations. The approach used in PETSc is to encapsulate mathematical algorithms using higher-level operations on parallel objects, which can be used in application codes written in Fortran, C, C++, Python, and Matlab. The library handles the detailed messages passing required during the coordination of the computation. All the communications are done using MPI. PETSc makes extensive use of the capabilities of MPI, exploiting persistent communication, communicator attributes, private communicators, and other features [66].

PETSc is built around a variety of data structures and algorithms objects that provide the building blocks for the implementation of large-scale application codes on parallel computers. The basic abstract data objects are index sets, vectors, and matrices. Each of this data abstractions has several dense and sparse representations and its elements can be distributed among several processes. For matrices, PETSc supports dense storage and CSR storage, as well as several specialized formats. Built on top of the data abstractions there are various classes of linear and nonlinear solvers, and time integrator objects.

OSKI

The Optimized Sparse Kernel Interface (OSKI) Library [127, 128] is a collection of low-level C primitives that provide automatically-tuned computational kernels on sparse matrices. OSKI provides basic kernels, like sparse matrix-vector multiply and sparse triangular solve. OSKI targets uniprocessors machines, although its developers are working in extending OSKI for vector architectures, SMPs, and distributed memory machines. Moreover, they are also working in integrating OSKI solvers with higher-level libraries such as PETSc.

Sparse structures in PGAS languages

Some works about the use of sparse structures in the PGAS languages exist in the literature. For example [57] evaluates the performance of different implementation of sparse data structures in UPC.

Chapel is the first PGAS language that had added support for spare arrays as a built-in feature. Chapel supports structures with different domain types including sparse arrays [26]. The Chapel sparse domains represent arbitrary subsets of indices of a bounding domain. Their arrays store an implicit zero value for any index that is within the parent domain but not within the child domain. The domain types support a rich set of operations including iteration, membership tests, and intersection.

2.6 Summary

In this chapter, we have reviewed a great number of parallel frameworks. Taking into account their execution model, they can be classify in the two categories we proposed: Data-mapping and data-flow.

In data-mapping, the data partition and mapping guide how the application works. The tasks are associated with memory portions, and communication and synchronization depend on their associated mapping. This can be seen in the solutions we discussed, such as OpenMP or PGAS languages, and static skeletons.

In data-flow, the data that flows between tasks is dynamically generated. The load balancing is done by a scheduler instead of being associated to the data partition. The data partition only affects the task granularity. Examples of this category are the stream frameworks and the dynamic skeletons.

Since our final objective is to create a unified parallel system, the runtime of that system has to support both execution models in order to allow the programmer to use different parallel programming paradigms. Moreover, it has to be able to manage sparse structures.

Chapter 3

The Hitmap library

Hitmap is a library for hierarchical tiling and mapping of dense arrays. It is based on a distributed SPMD (*Single Program, Multiple Data*) programming model, using abstractions to declare data structures with a global view, and automatizes the partition, mapping, and communication of hierarchies of tiles, while still delivering good performance.

In this Thesis, we use Hitmap as a base to create a runtime for a general parallel programming system. To extend Hitmap, we need first to understand its execution model and its current limitations. Thus, this chapter explores the details of the Hitmap library. It contains a description of the library features, architecture, and usage methodology. Finally, it also includes a complete case of study that helps to understand the usage of the library and its limitations, so it can be extended in following chapters.

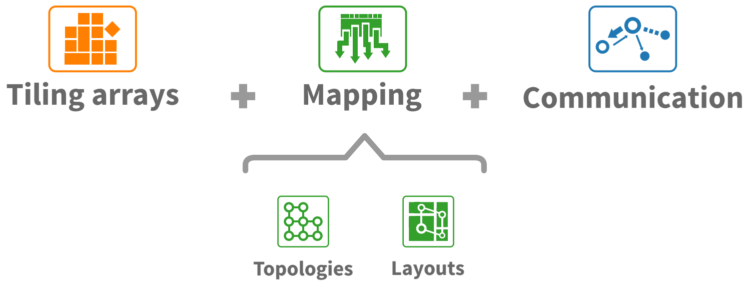


Figure 3.1: Hitmap library functionalities.

3.1 Hitmap features

Hitmap [59] has many features to perform hierarchical tiling and mapping of arrays. This library is designed to simplify the use of a global or local view of the parallel computation, allowing the creation, manipulation, distribution, and efficient communication of tiles and tile hierarchies.

In Hitmap, data-layout and load-balancing techniques are independent modules that belong to a plug-in system. The techniques are invoked from the code and applied at runtime when needed, using internal information of the target-system topology to distribute the data. The programmer does not need to reason in terms of the number of physical 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) Generate a virtual topology structure; (2) map the data grids to the different processors with chosen load-balancing techniques; (3) automatically determine inactive processors at any stage of the computation; (4) identify the neighbor processors to use in communications; and (5) build communication patterns to be reused across algorithm iterations. These functionalities are organized in three different categories: Tiling, Mapping, and Communications. The categories are represented in Fig. 3.1.

3.1.1 Terminology

Hitmap defines several concepts to write parallel programs. A *shape* represents the domain of the data used in the program while a *tile* is the entity that keeps the actual elements. The tiles are created using the shapes. In Hitmap, a *topology* describes the structure of the available processes. A *layout* is the entity that distributes the shapes onto a topology, dividing the domain for each element of the topology. The concept of *communication* represents data transmission between two or more processes. Finally, a *pattern* groups together several communications.

Tiling

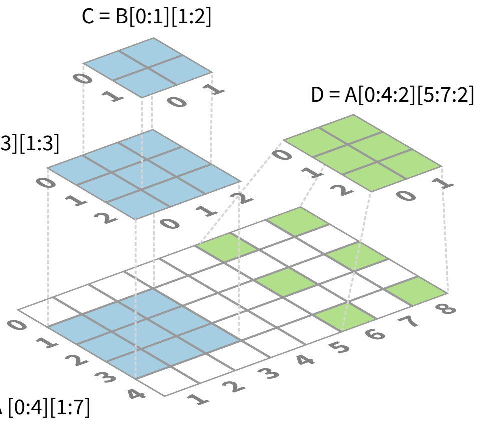


Figure 3.2: Tiling creation from an original array.

3.1.2 Tiling functionalities

The functionalities of this category allow to define and manipulate hierarchical array tiles. They can be used independently of the the rest of the library to improve locality in sequential code, as well as to manually generate data distributions for parallel execution. The tiles can be defined by the *domain* of an array using its particular range of indices. See array A in Fig. 3.2, where we use a notation to specify dimensions and ranges that resembles Fortran 90 and Matlab conventions. A tile can also be derived from another one, specifying a *subdomain*, which is a subset of the index ranges of the parent tile.

The elements of the original array can be accessed using two different coordinates systems, either the original coordinates of the array, or new tile coordinates starting at zero in all dimensions. See arrays B and C in Fig. 3.2, that are surrounded by their local coordinates indices. Tiles may also select subdomains with *stride*, transforming regular jumps in the original array indices to a compact representation in tile coordinates. See array D in Fig. 3.2.

The memory to store the elements is allocated on demand. Tiles may allocate own memory to store the array elements or they can reference the memory of their ancestors. Accesses to the elements are transparently mapped to the appropriate memory (tile memory or memory at the *nearest ancestor* with allocated memory). This tile allocation system greatly simplifies data-partition and parallel algorithm implementation. For example, the programmer may define a global array without allocated memory, and create derived tilings that will only allocate the needed local memory.

Hitmap tiles support many other operations, such as copies from/to ancestor/descendant tiles, overlapping tiles, extended tiles, or recursive updates.

3.1.3 Mapping functionalities

One of the key characteristics of Hitmap is that it clearly splits the mapping in two independent parts: topology and layout. Hitmap implements two separated and reusable plug-in systems: One to create virtual topologies of processors, and another for data distribution and layout functions.

The topology plug-ins create virtual topologies using internal data. Thus they hide the physical topology details. Its purpose is to move the reasoning in terms of physical processors from the programmer to the library. Current modules already available in Hitmap implement different topologies, such as grids of processors in several dimensions, or processor clustering depending on the sizes of data parts.

Data partition is done by the layout plug-ins. They automatically part an array into tiles, depending on the virtual topology selected. The partition modules receive a virtual topology, the layout function name to be used, and the data structure to be distributed. They return the part of the subdomain mapped to the local virtual process, and they can also be used to obtain information about other virtual processors parts.

In Fig. 3.3, topology functions have generated the requested topology using four processors. Given a topology, and the domain of a tile, the partition of that tile and their assignment to different processors are computed automatically. In the upper part of the figure, a one dimensional virtual topology is used with a *blocks* layout function. This results in a band partition. The lower part of Fig. 3.3 shows another example that uses a 2D topology, where *the same* layout function generates two dimensional blocks.

3.1.4 Communication functionalities

Hitmap supplies an abstraction to communicate selected pieces of hierarchical structures of tiles among virtual processors. It provides a full range of communication abstractions, including point-to-point communications, paired exchanges for neighbors, shifts along a virtual topology axis, collective communications, etc. The Hitmap approach encourages the use of neighborhood and tile information automatically computed by layouts to create communications which are automatically adapted to topology or data-partition changes. These communications can be composed in reusable patterns.

3.2 Hitmap architecture

This section describes the architecture of the Hitmap library. Hitmap was designed with an object-oriented approach, although it is implemented in the C programing language [83]. The classes are implemented as C data structures with associated functions. Figure 3.4 shows a UML class diagram of the library architecture. A summary of the basic Hitmap library API is shown in Table 3.1.

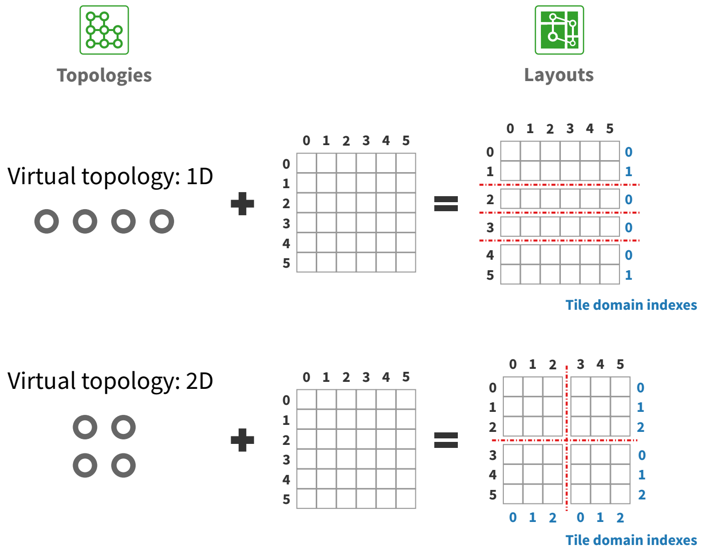


Figure 3.3: Different partitions automatically computed by a layout depending on the virtual processor topology.

The Hitmap domains are represented by the `Shape` class. A shape represents a subspace of array indices defined as an n-dimensional rectangular parallelotope. Its limits are determined by n `Signature` objects. Each signature is a tuple of three integer numbers $S = (b, e, s)$ (begin, end, and stride), representing the indices in one of the axis of the domain. Signatures with $s \neq 1$ define non-contiguous yet regular spaced indices on an axis. The index cardinality of a signature is $|S| = \lceil (e - b)/s \rceil$. Begin and stride members of a signature represent the coefficients of a linear function $f_S(x) = sx + b$. Applying the inverse linear function $f_S^{-1}(x)$ to the indices of a signature domain we obtain a compact, contiguous domain starting at $\vec{0}$. Thus, the index domain represented by a shape is equivalent (applying the inverse linear functions defined by its signatures) to the index domain of a traditional array.

A `Tile` object maps actual data elements to the index subspace defined by a shape. New allocated tiles internally use a contiguous block of memory to store data. Subsequent hierarchical subselections of a tile reference data of the ancestor tile, using the signature information to locate and access data efficiently. Tile subselections may be also allocated to have their own memory space.

The `Topology` and `Layout` abstract classes are interfaces for two different plug-in systems. These plug-ins are selected by name in the invocation of the constructor method. Programmers may include their own new techniques. Topology plug-ins implement simple functionalities to arrange physical processors in virtual topologies. Layout plug-ins implement methods to distribute a shape across the processors of a virtual topology. Hitmap has

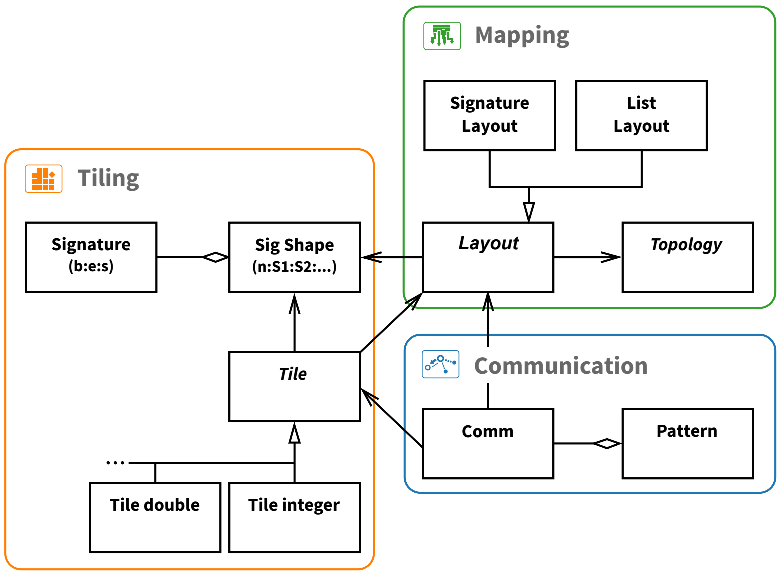


Figure 3.4: Hitmap library architecture.

Object	Method	Description
SigShape	`SigShape(dims, [begin,end,stride]*)`	Signature shape constructor. A SigShape is defined by a selection of indices in each of its dimensions.
Topology	`Topology(plugin)`	Topology constructor. This constructor creates a new topology object using the selected plugin to arrange the available processors in a virtual topology. It could be one of the Hitmap predefined plugins or a used-defined one.
Layout	`Layout(plugin, topology, shape)`	A Layout constructor determines the data distribution of a shape over a virtual topology. As in the topology objects, Hitmap offers several predefined plugins.
	`getShape([procId])`	This method returns the local shape assigned to a given processor, or the shape of the current processor.
Comm	`Comm(type, layout, TileIn, TileOut, [destination])`	Creates a new Comm object that communicates data from tiles using the processors involved in the layout. The type parameter define the kind of communication to be performed.
	`do()`	Performs the communication encapsulated by the Comm object.
Pattern	`Pattern(type)`	Creates a new communication pattern, which can be executed ordered or unordered.
	`add(comm)`	Adds a new communication to the pattern.
	`do()`	Performs the communications of the pattern.
Tile	`Tile(shape, datatype)`	Tile constructor. It creates a tile object using the domain defined by a shape object. The datatype parameter determines the type of data of the elements.
	`allocate()`	Allocates memory for the tile.
	`elemAt(x,y,...)`	Method to access an element in the tile.
	`select(dims,[begin,end]*)`	Method to create a subtile.

Table 3.1: Classes and methods of the Hitmap API.

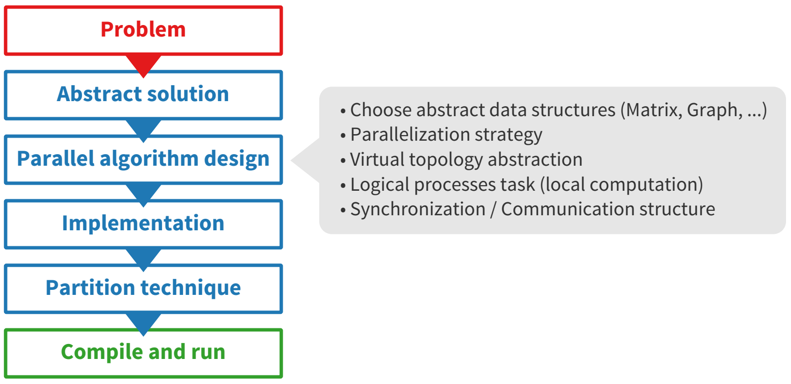

Problem
Abstract solution
Parallel algorithm design
Implementation
Partition technique
Compile and run
• Choose abstract data structures (Matrix, Graph, ...)
• Parallelization strategy
• Virtual topology abstraction
• Logical processes task (local computation)
• Synchronization / Communication structure

Figure 3.5: Hitmap programming methodology.

different partitioning and load-balancing techniques implemented as layout plug-ins. They encapsulate details which are usually hardwired in the code by the programmer, improving reusability. The resulting layout object contains information about the local part of the domain, neighborhood relationships, and methods to locate the other subdomains.

Finally, the `Comm` (communication) class represents information to synchronize or communicate tiles among processors. The class provides multiple constructor methods to build different communication schemes, in terms of tile domains, layout objects information, and neighbor rules if needed. The library is built on top of the MPI communication library, for portability across different architectures. Hitmap internally exploits several MPI techniques that increase performance, such as MPI derived data-types, and asynchronous communications. Communication objects can be grouped using the `Pattern` class. Objects of this class are reusable patterns to perform several related communications with a single call.

3.3 Hitmap usage methodology

In this section, we discuss how a typical parallel program is developed using Hitmap. Hitmap proposes a programming methodology that follows a waterfall model with the phases shown in Fig. 3.5. Decisions taken at any phase only affect subsequent phases.

3.3.1 Design of a parallel program using Hitmap

The programmer designs the parallel code in terms of logical processes, using local parts of abstract data structures, and exchanging information across a virtual topology of unknown size. The first step is to select the type of virtual topology appropriate for the particular parallel algorithm. For example, it could be a rectangular topology where processors have two indices (x, y). Topologies define neighborhood relationships.

The second design step is to define domains, starting with a global view approach. All logical processes declare the shapes of the whole data structures used by the global computation. The programmer chooses where to activate the partition and mapping procedure for each domain. At this phase, the only information needed is the domain to be mapped. There is no need to specify the partitioning technique to be used. Local domains may be expanded to overlap other processors subdomains, generating *ghost zones*, a portion of the subspace shared (but not synchronized) with another virtual processor. Once mapped, and after the corresponding memory allocation, the programmer can start to use data in the local subdomain.

The programmer finally decides which communication structures are needed to synchronize data between the computational phases. They are imposed by the parallel algorithm. At this phase, the programmer reasons in terms of tile domains and domain intersections.

3.3.2 Implementation of a parallel program using Hitmap

Hitmap provides functionalities to directly translate the design to an implementation which is independent of the underlying physical topology, and the partition/layout techniques. Hitmap provides several topology plug-ins. Each plug-in automatically arranges the physical processors, using its own rules to build neighborhood relationships. New topology plug-ins with different rules can be developed and reused for other programs. Topology plug-ins may flag some processors as inactive transparently to the programmer. For example, when there are more processors than gaps in the virtual topology.

Global domains are declared with shape objects. Layout objects are instantiated for partitioning and mapping global domains across the virtual topology. The layout objects are queried to obtain the local subdomain shapes. After expanding or manipulating them if needed, the local tiles can be dynamically allocated. Once allocated, data in the local tiles may be accessed using local tile coordinates, or in terms of the original, global view coordinates. This helps to implement the sequential computations for the tiles.

Communication objects are built to create the designed communication structures. They are instantiated using a local tile (to locate the data in memory) and using information contained in a layout object about neighbors and domain partition. For example, for ghost zones, shape intersection functionalities automatically determine the exact chunks of data that should be synchronized across processors. The communication objects contain data-marshalling information. They can be created at program initialization, and invoked when they are needed, as many times as required.

The result of the implementation phase is a generic code that is adapted at runtime depending on: (a) The particular global domains declared; (b) the internal information about the physical topology; and (c) the selected partition/layout plug-ins. Note that it is possible to change the partition technique just by selecting it by name, without affecting the rest of the code.

3.4 Hitmap case study

This section contains a Hitmap case study of a real parallel application for dense arrays. It is a motivational example previous to the work carried out in this Thesis. With it, we will show how to design and implement an application with Hitmap, and how to extend the library using the plug-in system. This example help us to understand the Hitmap execution model, and to find its limitations so we can solve them.

The chosen example is the well-known multigrid MG NAS parallel benchmark [12]. This benchmark is a V-cycle multigrid kernel, which solves a 3D PDE (*Partial Differential Equation*). The contributions of the PhD candidate for this example include the review of multigrid methods to understand the algorithm, the study of the different MG implementations available, the implementation of the MG benchmark using the Hitmap library, and the creation of new data-layout that generalizes the data-alignment policy of the benchmark, in order to automatically adapt the data-distribution and communication code to any grain level.

These contributions have been published in [47, 59]. These papers show that it is possible to introduce flexible automatic data-layout techniques in current parallel compiler technology without sacrificing performance and greatly reducing the development effort when comparing with coding these details manually.

3.4.1 MG Benchmark

The NAS MG Benchmark is a structured V-cycle multigrid kernel to solve the Poisson equation for heat transfer in a discretized 3D space, using an iterative Jacobi solver. MG is part of the *NASA Advanced Supercomputing* (NAS) benchmarks [12]. These benchmarks are well-known and widely-used applications to compare performance of parallel computers.

Sequential algorithm

The NAS MG benchmark performs iterations of a V-cycle multigrid algorithm to solve the Poisson equation $\nabla^2 u = v$ on a cubical grid with periodic boundary conditions [11]. Each iteration consists of an evaluation of the residual (r), and the application of the correction:

$$\begin{aligned} r &= v - Au \\ u &= u + M^k r \end{aligned}$$

where A is the trilinear finite element discretization of the Laplacian ∇^2; M is the V-cycle multigrid operator; and $k = log_2(d)$, where d is the grid dimension size.

In Fig. 3.6, we show the algorithm applied at each level, and a representation of how the operators are linked through the multigrid levels. The four operators represented in the figure are: P (Restriction), Q (Prolongation), A (Evaluation), and S (Smoother). All these operators are implemented as 27 coefficients arranged as a $3 \times 3 \times 3$ cubical array that is applied as a stencil operator, as function of the values obtained in the previous iteration in the neighbor array cells.

Parallel algorithm

The MG code distributes the data across p processors, where p is a power of two. It divides the original grid in contiguous blocks. Since in each level there is a different grid, all processor will have a set of blocks, one for each level. The processors are arranged in a 3D topology. It does not need to be perfectly cubic, but in each dimension the number of processors must be power of two.

Because both, the number of processors in any dimension and the grid sizes are defined as powers of two, all the processors have an equal block size. However, when there are more processors than data to spread (on the coarser grids), some processors will not have any elements to process, and they will become inactive during the computation of that level.

To perform the operations in the border elements of the block, each processor also needs part of the data assigned to its neighbors. This data is usually called *ghost zone* or *ghost elements*. We name *border elements* to the area immediately inside the boundary of each block. Therefore, the ghost elements are border elements in the neighbors.

Figure 3.7 shows the application of the Q operator in one dimensional axis transitioning from levels where there is more processors than data in that axis. In the left side of the figure, we can see the layout for a sequential multigrid with three levels. In the right side, we can see the distribution in blocks. The light squares are the ghost elements. After calculating the residual, smoother, or restriction operations, each process needs to rebuild the ghost zone with the values updated in another processor. Thus, they need to exchange their border elements with the neighboring processors.

At the bottom level (coarse grid) there are not enough data for all processors, and processor 1 and 3 remain inactive. The neighborhood relationship changes because it is necessary to skip the inactive processors during the border elements exchange. Moreover, when coming back in the V-cycle, the reactivated processors have no elements. They receive two elements from the left neighbor and only one element from the right neighbor. Thus, the communication patters are different for some processors at given levels.

3.4.2 Implementations

The reference code of the MG NAS benchmark is written in Fortran, using the MPI message-passing interface for synchronization and communication. In this section we describe three different implementations: The NAS MG original code, a manually optimized translation to C language, and an implementation that uses Hitmap.

```
$z_k = M^k r_k$:
  if $k > 1$:
    $r_{k-1}$    $= P r_k$                  (Restrict residual)
    $z_{k-1}$    $= M^{k-1} r_{k-1}$        (Recursive solve)
    $z_k$        $= Q z_{k-1}$              (Prolongate)
    $r_k$        $= r_k - A z_k$            (Evaluate residual)
    $z_k$        $= z_k + S r_k$            (Apply smoother)
  else:
    $z_1$        $= S r_1$                  (Apply smoother)
```

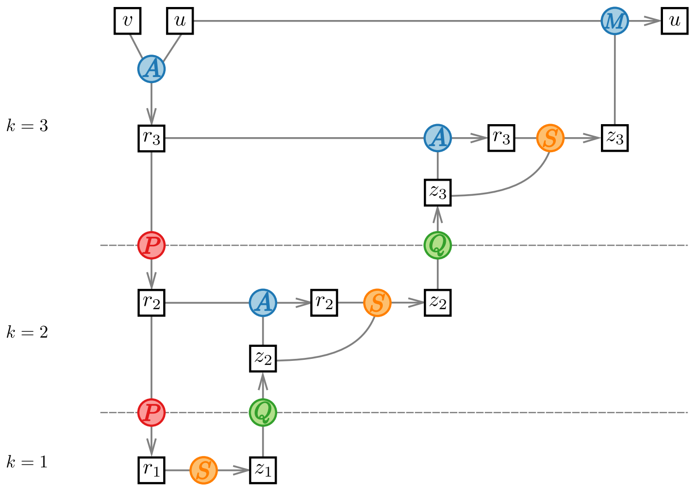


Figure 3.6: MG algorithm and multigrid operations.

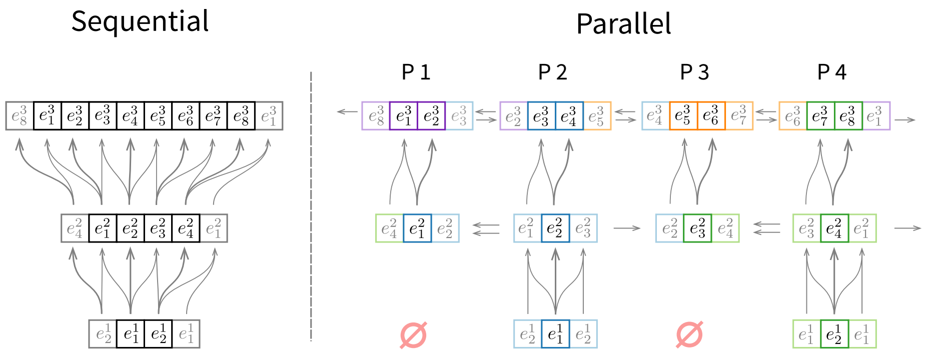


Figure 3.7: Application of Q operator for a partition on a given grid dimension.

NAS implementation

The NAS MG reference code uses some implementation and optimization techniques to increase performance. The most simple one is to skip the calculations of null coefficients in the operators. Moreover, it uses partial sums when applying the operators to avoid repeating the same calculations and to exploit cache memory. Finally, it does a reordering of the prolongation code that allows a fully overlap of the u and z array at the top level resulting in a reduction of operations and memory usage.

We have found further room for improvement that can be applied to the NAS Fortran reference code. It is possible to avoid cleaning communication buffers before receiving data. This improvement has a noticeable impact in terms of benchmark performance. Our experimental results show that this unnecessary operation slows down the overall performance up to 20%.

Efficient C implementation

The NAS MG implementation using MPI is written in Fortran, while the Hitmap library is written in C. In order to better compare the codes, we have manually translated the NAS MG benchmark to C language. To create this version, we have used the C-OpenMP version of the NAS benchmarks as the starting point, including the MPI communication structure of the original Fortran version. We have taken into account the particularities of both languages, such as differences on function interfaces semantics, array indices realignment, and storage of data structures.

Hitmap implementation

The Hitmap implementation uses the main computation and other sequential parts of the direct C translation, adapting them to work with Hitmap tiles. Data-layout and communications have been generated directly using Hitmap calls. In this section we discuss the Hitmap techniques needed to automatically compute the data-layout and exploit reusable communication patterns in the MG code.

In Listing 3.1, we show an excerpt of the Hitmap code executed at the start of the application to precompute the data layout and communication patterns. Line 2 creates a 3-D virtual topology of processors, using the internal information of the real topology obtained by the low-level layer (in MPI, the local identifier and the whole number of physical processors).

The rest of the code is executed in a loop for each grid level k. First, a shape which defines the whole grid is constructed (Lines 5 to 7). The data-layout information is generated automatically with only one Hitmap function call (Line 10). The layout parameters are: (a) The layout plug-in name, (b) a virtual topology of processors; and (c) a shape with the domain to distribute. The result is a `HitLayout` object, containing the shape assigned to the local processor and neighbors information. The function in Line 11 simply adds the circular shift property to the neighborhood relationship. The layout function is applied to all dimensions of the input shape by default. An optional parameter may indicate the application of the layout function to only a given dimension. Specific plug-in modules may define extra parameters. We further discuss the plug-in module for the MG benchmark below.

In Line 14, the domain shape of the local block is obtained from the layout object, and expanded by two elements on every dimension to contain the ghost zones. This shape is used to declare and allocate the local block as a tile of double elements (Line 18). All the communications needed for border exchange, and inactive processor reactivation, are encapsulated using Hitmap communication patterns. In Line 22 of Listing 3.1, a new empty communication pattern is defined, whose communication actions will be executed in order. Finally, Lines 25 to 43 add communication actions to the pattern for this local processor. For each dimension, we determine the neighbors and we create a shape to select which part of the tile is sent or received. Since the layout object contains all the information about shapes, active processors, and neighborhoods, the code between Lines 25 to 43 is common to both, active and inactive processors, at any level of the multigrid computation. Data marshaling and unmarshaling is automatized by the actions of the communication patterns. When a processor needs to exchange the borders, it simply executes the appropriate pattern for the level with a single Hitmap function call whose only parameter is the pattern (not shown in the figure). The patterns are reusable through all the algorithm iterations.

Finally, it is worthwhile to note that, in NAS MG, array accesses are done calculating array indices manually on a flattened array. In the Hitmap version, accesses are simplified from the programming point of view through the use of tile access macros.

Building a layout plug-in

The extensible Hitmap plug-in system allows the programmer to create new topology and layout modules. MG needs a blocking layout with an application-defined policy to determine inactive processors on the coarsest levels.

```
/* 3D Array Topology */
HitTopology topology = hit_topology(plug_topMesh3D);

/* Create shape for global cubic grid */
grid_size_k = GRID_SIZE / pow(2, k);
HitSig sig = hit_sig(0, grid_size_k - 1, 1);
HitShape grid = hit_shape(3, sig, sig, sig);

/* Layout */
HitLayout layout = hit_layout(plug_layBlocksL, topology, grid);
hit_layWrapNeighbors(&layout);

/* My block domain */
HitShape block_shape = hit_shapeExpand(hit_layShape(layout) , 3, 1);

/* Allocate blocks */
HitTile_double block_v;
hit_tileDomainShapeAlloc(&block_v,sizeof(double),HIT_NONHIERARCHICAL,block_shape);
...

/* Compute communication patterns for border exchange */
HitPattern pattern_v = hit_pattern(HIT_PAT_ORDERED);
...

for(dim = 0; dim < 3; dim++){

   /* Get neighbor location */
   HitRanks nbr_l = hit_layNeighbor(layout, dim, -1);
   HitRanks nbr_r = hit_layNeighbor(layout, dim, +1);

   /* Shape for the face to Give and to Take */
   HitShape faceGive_l = hit_shapeBorder(block_shape, dim, HIT_SHAPE_BEGIN, 0);
   HitShape faceTake_r = hit_shapeBorder(block_shape, dim, HIT_SHAPE_END, 1);
   faceGive_l = expand_shape(faceGive_l, dim, 1);
   faceTake_r = expand_shape(faceTake_r, dim, 1);

   /* Add communication actions to pattern */
   hit_patternAdd(&pattern_v,
      hit_comSendRecvSelectTag(layout, nbr_l, &block_v, faceGive_l,
      HIT_COM_ARRAYCOORDS, nbr_r, &block_v, faceTake_r,
      HIT_COM_ARRAYCOORDS, HIT_DOUBLE, tag(0)));
   ...
}
```

Listing 3.1: Construction of the data layout and communication patterns for MG.

MG distributes the data in blocks. The operations are designed to keep maximum locality on coarse grids when data elements are on the right, and inactive processors on the left, for each group of processors. At bottom level in Fig. 3.7, we can see an example where there are only two data elements to distribute among four processors. The layout module should create two groups (1-2 and 3-4) but only processors 2 and 4 will have data elements to work with.

We have built a new layout plug-in for the library named `BlocksL` (Blocks with active leader at the Last processor of the group). A new plug-in is defined by two functions, one to calculate the proper slicing, and another one to define the neighborhood relationship (skipping possible inactive processors), for one dimension domain. The plug-in system systematically applies these functions to the required dimensions.

Let P be the number of virtual processors on a given dimension, p the index of the local processor, and D the number of domain indices to be distributed on the same dimension. Let $S = (b, e, s)$ be the signature expressing the range of domain indices to distribute, from b to e with stride s. Active processors are characterized by the following logical expression:

$$\lfloor p \times B/P \rfloor \neq \lfloor (1+p) \times B/P \rfloor$$

Let $a = 1$ if $B > P$, and $a = 0$ otherwise. For the active processors, the signature that defines their local domain indices is $S' = (b', e', s')$, where

$$b' = \lfloor p \times B/P \rfloor \times s + b$$

$$e' = ((p+a) \times \lfloor B/P \rfloor - a) \times s + b$$

$$s' = s$$

The implementation of the function that calculates the local block signatures for a given dimension is shown in Listing 3.2. It receives four parameters: (1) The processor index in the dimension; (2) the number of processors in the dimension; (3) the signature with the domain to distribute; and (4) a pointer to a signature to return the result. The function calculates the begin, end, and stride that define the local block on the given dimension, returning `TRUE` if the local processor is active, and `FALSE` otherwise.

More sophisticated partition modules may define extra parameters. For example, some modules receive an estimation of the load associated with the domain indices, which may be used to generate an adaptive load-balance, or to decline further parallelization when the grain is too fine.

The function shown in Listing 3.3 defines a specific dimensional neighborhood relationship that skips inactive processors for the `BlocksL` layout. This function receives five parameters related to a given dimension: (1) The processor index; (2) the number of processors; (3) the domain cardinality; (4) the distance to the neighbor in that axis; and (5) a flag that indicates whether wrapping is active for this dimension. The function returns the corresponding neighbor coordinate for this virtual topology.

```c
int hit_layout_plug_layBlocksL_Sig(int procId,int procsCard,HitSig in,HitSig *out){

   int blocksCard = hit_sigCard( in );
   double ratio = (double)blocksCard / procsCard;
   double beginFrac = procId * ratio;

   /* Detect Non-Active virtual process (not enough logical processes) */
   if ( floor(beginFrac) == floor(beginFrac+ratio) ) {
      (*out) = HIT_SIG_NULL;
      return FALSE;
   }
   else {
      /* Compute signature */
      (*out).begin = (int)beginFrac * in.stride + in.begin;
      int adjust = (blocksCard > procsCard) ? 1 : 0;
      (*out).end = (((procId+adjust) * (int)ratio)-adjust) * in.stride + in.begin;
      (*out).stride = in.stride;
      return TRUE;
   }
}
```

Listing 3.2: Plug-in function to generate a blocking partition compatible with MG, for any dimension.

```c
int hit_layout_plug_layBlocksL_neighbor(int procId, int procsCard, int blocksCard,
   int shift, int wrap) {

   /* Compute active neighbor */
   double ratio = (double) blocksCard / procsCard;
   if(ratio > 1) ratio = 1;
   int neighIdAct = (int)(procId * ratio) + shift;
   int activeProcs = min(procsCard,blocksCard);

   /* Check if the neighbor is out of the processors range: apply wrapping */
   if ( neighIdAct < 0 || neighIdAct >= activeProcs )
      if(wrap != HIT_WRAPPED) return HIT_RANK_NULL;
      else neighIdAct = ((neighIdAct % activeProcs) + activeProcs) % activeProcs;

   /* Return the last processor of the group (located before the next group) */
   return (int) ceil((neighIdAct+1) * 1/ratio) -1;
}
```

Listing 3.3: Plug-in function to handle neighborhood relationships compatible with MG.

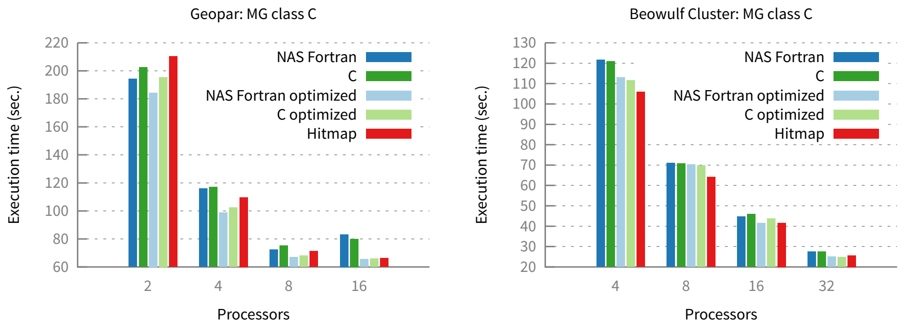


Figure 3.8: Comparison of execution times between NAS Fortran, C manually optimized, Hitmap, and the optimized versions.

To keep generality of application, all data-layout plug-ins should work properly for any number of processors P and any domain cardinality B. Their functions should perform complete mappings when $P \leq B$, and inactive processors should be identified and skipped when $P > B$. Thus, this plug-in generalizes the MG partition policy, eliminating any topology or grid dimension restrictions found in the original implementations. Moreover, it represents a generic pattern that is also reusable in other parallel algorithms, such as cellular automata, block matrix multiplication, or image filtering.

Testing combinations of topology vs. layout functions is really easy, changing only the names or parameters of the topology or layout functions. For example, we may use a *mesh1D* topology plug-in to create a one dimensional collection of coarse 3D blocks, or use an extra parameter to restrict the layout to specific dimensions, creating different band-based partitions instead of 3D blocks.

3.4.3 Experimental results

In this section we compare the three MG versions described above, both in terms of performance and programmability. The codes have been run on two different architectures. The first one, Geopar, is an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32Gb of RAM. Geopar runs OpenSolaris 2008.05, with the Sun Studio 12 compiler suite. The second architecture is an homogeneous Beowulf cluster of up to 36 Intel Pentium IV nodes, interconnected by a 100Mbit Ethernet network. The MPI implementation used in both architectures is MPICH2, compiled with a backend that exploits shared memory for communications if available in the target system.

Due to the memory size restrictions on the Beowulf cluster nodes, the experiments have been run using the MG C class problem, that performs 20 iterations in a 512^3 sized grid. Following NAS practices, performance results were recorded skipping initialization time.

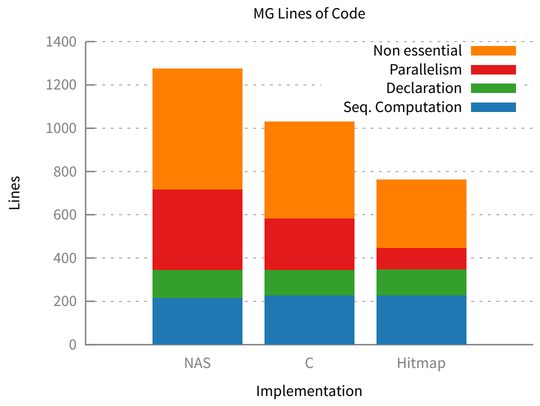


Figure 3.9: Comparison of the number of code lines.

We also show the performance of two additional versions: "NAS Fortran Optimized", consisting of the original NAS code but without the unnecessary clean of the reception buffers (see Sect. 3.4.2), and "C optimized", that consists of the C version of this optimized code. These versions were developed to isolate the effect of this optimization, since Hitmap does not need to clean reception buffers.

Figure 3.8 shows the execution time of the main computation part for these five versions. In the Geopar system, all versions surpass the original NAS code. As expected, the effect of the optimized versions is noticeable in terms of performance. Hitmap uses complex macro functions to expose to the native C compiler the formula used to access tile elements. The optimizations obtained with the Solaris compiler are not as good as with GCC. Thus, the sequential time on the Geopar machine leads to slightly higher execution times for the Hitmap version using few processors. Nevertheless, Hitmap version scales better than the others due to its reduced communication costs. In the Beowulf cluster (Fig. 3.8), the use of Hitmap leads to even better performance figures, since communications are more costly in this architecture, and Hitmap takes advantage of several MPI advanced capabilities automatically. For example, the Hitmap communication functions precalculate hierarchical MPI derived data types for each tile. The resulting objects are efficiently reused on each iteration for faster marshalling and unmarshalling. The beneficial effects of using appropriate MPI derived data types for communications in terms of performance has been reported in [87].

Finally, Fig. 3.9 shows a comparison of the Hitmap version with the Fortran and the manually optimized C versions in terms of lines of code. We distinguish lines devoted to sequential computation, declarations, parallelism (data layouts and communications), and other non-essential lines (input-output, etc). Note that the internal lines of code of the Hitmap library are not included in the comparison. The codes of Listing 3.2 and Listing 3.3, which have 37 additional lines, are excluded as well.

Taking into account only essential lines, our results show that the use of the Hitmap library leads to a 37.4% reduction on the total number of code lines with respect to the Fortran version, and a 23.2% reduction with respect to the C version. Regarding lines devoted specifically to parallelism, the percentages of reduction are 72.7% and 57.4%, respectively.

The reason for the complexity reduction in the Hitmap version is that Hitmap greatly simplifies the programmer effort for data distribution and communication, compared with the equivalent code needed to manually calculate the information needed to be used in the MPI routines. In particular, Hitmap avoids the use of tailored formula to compute local tile sizes, mapping of data to virtual processors, and neighborhood relationship, at the different grain levels. Moreover, the use of Hitmap tile-management primitives eliminates some more lines in the sequential treatment of matrices and blocks.

3.5 Limitations of Hitmap

As we have shown in the previous sections, Hitmap tiling and mapping functionalities make it easy to develop parallel code. Using the combination of tile, topology, layout, and communication objects, it is possible to develop a code that adapts automatically to the problem size in each level of the multigrid. Hitmap works well for this example. However, for other kind of parallel programming, it has several limitations:

- Hitmap focus on dense domains and regular subselections. The tile objects are multidimensional arrays that can be created selecting portions of previous tiles. Although, it is possible to implement complex data distributions and communications, Hitmap has no direct support for other common types of data structures, such as sparse matrices or graphs.

- All the automatic functionalities in Hitmap are dependent on the data mapping and layout. This means that it is very useful when developing applications where the data-mapping drives the computation. But, for task-parallel or dynamic applications, it does not offer any additional features than a regular message passing library.

3.6 Conclusions

In this chapter, we have introduced Hitmap, a runtime library for hierarchical tiling and mapping of dense arrays. We have discussed its features, the design of its architecture, its usage methodology, and we have presented a real application as a case study.

The programming approach used in Hitmap can be used to generalize and encapsulate data-partition policies, providing a higher degree of application control and flexibility due to its decoupled mapping system, based on two types of combinable plug-ins. Our results [47, 59] show that it is possible to introduce flexible automatic data-layout techniques in current parallel compiler technology, greatly reducing the development effort when comparing with coding these details manually, without sacrificing performance.

In the following chapters, we will use Hitmap library as a starting point to fix its limitations and to extend it with sparse data management, and a new dataflow model as runtime support.

Chapter 4

Unified support for dense and sparse data

Dealing with both dense and sparse data in parallel environments usually leads to one of two different approaches: To rely on a monolithic, hard-to-modify parallel library; or to code all the data management details by hand. For our goal of creating a runtime system for generic parallel framework, the runtime has to integrate dense and sparse data management using a common interface whenever possible. We propose a solution that decouples data representation, partitioning, and layout from the algorithmic and from the parallel strategy decisions of the programmer. We will show that this approach delivers good performance while the underlying library structure remains modular and extensible.

4.1 Introduction

Data structures with sparse domains arise in many real problems within the scientific and engineering fields. For example, they appear in PDE solvers as sparse matrices, or they are used to model complex structural element relationships as sparse graphs. However, not many parallel programming frameworks transparently integrate support for both dense and sparse data structures. Most parallel programming languages, such as HPF [82] or UPC [23], only have a native support for dense arrays, including primitives and tools to deal with data locality and/or distribution only for dense data structures. Coding sparse-oriented applications with these languages implies manually managing the sparse data with an expensive programming effort, or using domain specific libraries that do not follow the same conceptual approach. In both cases, the reusability in the sparse domain of code developed previously for dense data structures is very poor.

In this chapter, we present a complete solution to handle sparse and dense data domains using the same conceptual approach. The rationale is that the design of a parallel algorithm follows the same basics steps regardless of the underlying data structure; the differences appear at the implementation stage, when specific methods for data partition and distribution should be selected. Thus, the implementation of the algorithms could be simplified using high-level parallel models with abstractions for the data distribution management. Providing the appropriate abstractions for the data storage, partition, layout, and communication structure building, it is possible to reuse the same parallel code independently of the domain representation and the particular partition technique selected.

To validate our solution, we have integrated sparse and dense support into the Hitmap library. The resulting library supports dense and sparse matrices and graphs, and can be extended to support other structures as well. With our solution, it is possible to build parallel programs in terms of an explicit, yet abstract, synchronization and communication structure, that automatically adapts to efficiently use both dense and sparse data domains.

To show the application of this approach, we have tested different benchmarks. Our experimental results show that the use of our solution greatly reduces the associated programming effort while keeping a performance comparable to those obtained by manual programming, or using well-know frameworks.

4.2 Conceptual approach

In this section, we discuss how a common interface for sparse and dense data management, combined with abstractions to encapsulate the partition, layout, and communication techniques, lead to an unique parallel programming methodology.

Regardless the dense or sparse nature of the data, most parallel programs follow the same strategy, with well-known stages. Figure 4.1 shows these stages for different data

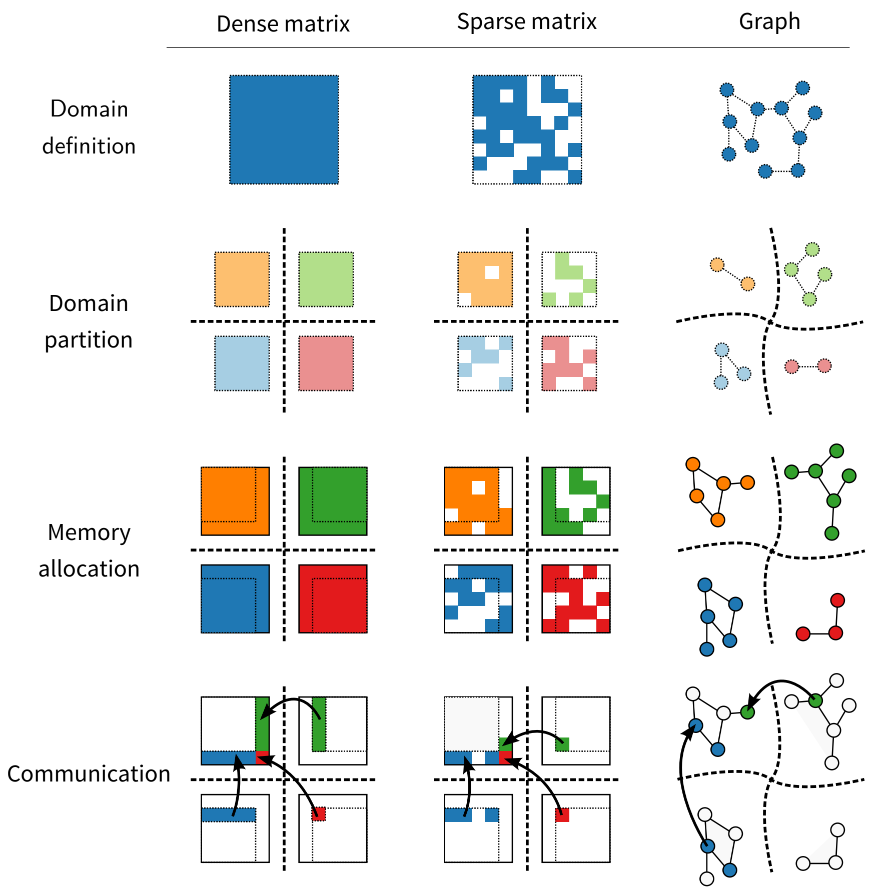


Figure 4.1: Initialization stages of parallel computations for different data structures.

domains. As we will see, the stages are conceptually independent form the data domain, the partition technique, and the specific algorithm applied to the data. We will use several abstract mathematical entities to describe these stages involved in the design of most parallel programs. In the following sections, we will describe how these entities are implemented in the Hitmap library and how they can be used.

Domain definition The first stage is to define the data domain. We define a *domain* D as a collection of n-tuples of integer numbers that define a space of n-dimensional indices. For dense arrays, the index domain is a subspace of Z^n, defined by a rectangular parallelotope. For sparse data structures it is just a subset of Z^n.

Domain partition The next stage is the domain partition. It consists in dividing a whole data domain into smaller portions, assigning them to different processors. The processors are arranged using a *virtual topology* of processors V. The virtual topology defines neighborhood relationships. There are many algorithms and methods to perform a partition. The result of the partition is a set of domain subspaces containing the local elements for each processor. The particular partition method can be calculated using a *layout* function L, which maps the domains subspaces to the processors in a selected virtual topology $L(D, V) : D \to V$.

Memory allocation Once the domain has been partitioned, each processor has to allocate memory for the elements of its local subspace. We define a *tile* T as an object that associates data elements to index elements of a domain D. *Matrix tiles* associate one data element to each domain element. *Graph tiles* are defined for 2-dimensional domains. Each domain element $(i, j) \in D$ indicates the existence of a graph edge. A graph tile associates one data element to each domain element (edge values), and one data element to each single index $i : (i, j) \in D \vee (j, i) \in D$ (node values). The processors need memory to keep the values of their local elements. They may also allocate additional memory for elements mapped to other processors that are needed to complete the local computation. Tiles can be created and allocated using the local part of the domain assigned by the layout function. Note that buffers for neighbor data can be automatically derived.

Communication Communication structures can be defined as abstract objects in terms of neighborhood relationships. If tiles are used to allocate memory for elements mapped to other processors, it is also possible to define communications in terms of the overlappings of local and neighbor tile domains.

Local computation After these four preliminary stages, local computations are carried out using iterators to retrieve the data from the tiles.

To sum up, the presented strategy focus on two aspects: Abstract data management, and explicit communications. The domain definition, data allocation, and layout are encapsulated using abstractions. Thus, they are independent of the used data type, dense or sparse. With the previous abstractions for data management is it possible to define abstract explicit communications.

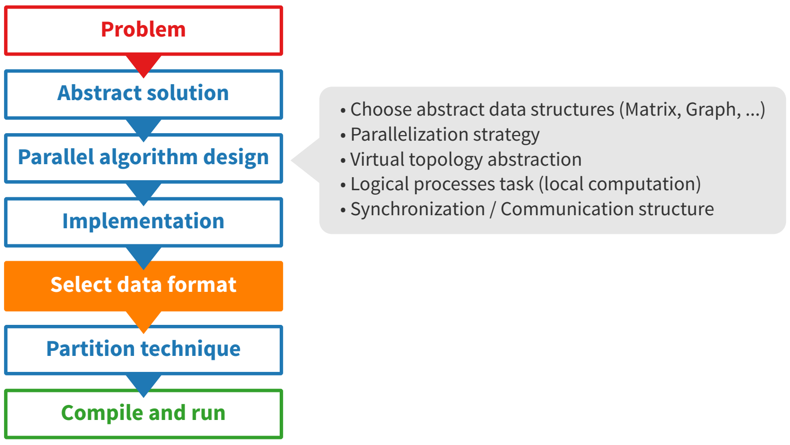

Problem
Abstract solution
Parallel algorithm design
Implementation
Select data format
Partition technique
Compile and run
• Choose abstract data structures (Matrix, Graph, ...)
• Parallelization strategy
• Virtual topology abstraction
• Logical processes task (local computation)
• Synchronization / Communication structure

Figure 4.2: Revised Hitmap programming methodology. The highlighted box represent the new decision phase introduced.

4.3 Adding support for sparse domains to Hitmap

In this section, we discuss simple abstractions to use the same methodology to manipulate dense and sparse data structures. We analyze the conceptual and design changes required in the Hitmap architecture to integrate sparse domains and their completely different partition techniques. Decoupling shapes and tiles from partition techniques and derived communications allows, for example, to extend the library with new data-structure classes without the need of reimplementing existing codes that carry out the computation.

Figure 4.2 shows the Hitmap programming methodology that was first introduced in Chapter 3, updated to include a new step. The new highlighted box represents the new decision phase that we have added to the Hitmap programming approach.

We have changed the design of the architecture, modifying previous classes and including new ones to support the sparse domains. An updated UML diagram is shown in Fig. 4.3. The new classes and methods of the Hitmap API are summarized in Table 4.1.

The original Hitmap library exploited the idea of separating the array index domain (in the Shape class) from the data allocation (in the Tile class). The original `Tile` class contained the methods to associate each multi-dimensional index with one data element, using linear functions defined by the signature objects in the shape. We extend this idea, transforming the `Shape` and `Tile` classes in abstract interfaces that can be implemented in different ways.

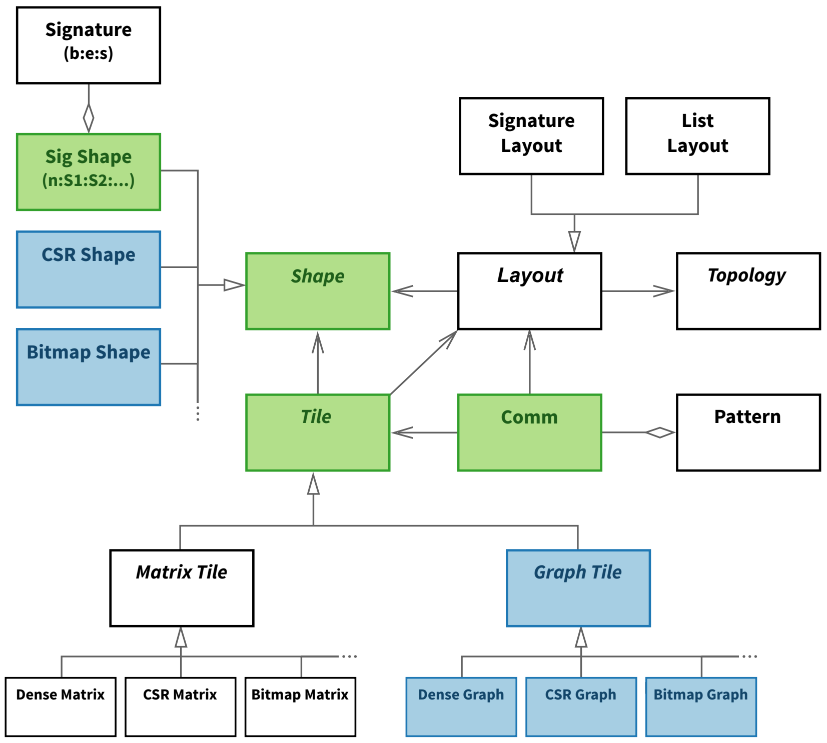


Figure 4.3: New Hitmap library architecture. Blue boxes represent the new classes introduced to integrate dense and sparse data support. Green boxes represent classes that had to be redesigned to deal with the new abstractions.

Object	Method	Description
SparseShape	SparseShape()	Sparse Shape constructor.
	addVertex(x)	Adds a new vertex to the structure.
	addElem(x,y), addEdge(x,y)	Functionally-equivalent methods to add an element to the matrix, or an edge to the graph.
	hasVertex(x)	Checks whether a vertex is present in the sparse structure.
	hasElem(x,y), hasEdge(x,y)	Functionally-equivalent methods to check whether an element is present in the matrix, or an edge in the graph.
	expand(SparseShape original, int dst)	Method to expand the current shape with the vertices of the original shape that are at a given distance (dst).
	vertexIterator(var,shape)	The vertex iterator modifies the variable var with the index of the vertices of the given shape.
GraphTile	GraphTile (Shape, datatype, vertex/edges)	GraphTile constructor. It creates a graph tile object using the domain defined by a shape object. The datatype parameter determines the type of data of each single element. The third argument is a flag to indicate whether the GraphTile object stores data for the vertex and/or edges of a graph.
	allocate()	Allocates the data for the tile.
	vertexAt(x)	Method to access a vertex in the tile.
	edgeAt(x,y)	Method to access an edge in the tile. Equivalent to elemAt(x,y) of Tile class.

Table 4.1: New classes and methods of the Hitmap API to handle sparse domains.

4.3.1 Shapes

The original `Shape` class is substituted by an abstract interface. The new `Shape` interface defines methods to create multi-dimensional index domains, add new elements to the domain, checks if an element is inside the domain, etc. The old `Shape` class, based on signatures, is transformed into a different class which implements the new `Shape` interface.

Sparse domains can be implemented in different ways, most of them related to traditional sparse matrix formats (COO, CSR, etc.). Some formats lead to the same shape implementation, because the methods to retrieve or locate data are not in the `Shape` interface. They will be found in the data-localization functions of the tile. The old signature shape implementation in Hitmap (`SigShape`) was very efficient to locate dense information, but did not directly support a proper representation for dynamically adding or eliminating particular indices. To solve this problem, we propose new implementations of the `Shape` class that are efficient enough to locate and traverse highly dense structures, but also allows us to represent particular holes in the index domain.

We have included classes for two new kinds of sparse domains:

- **CSR** As an example of traditional sparse domain representations, we have selected the *Compressed Sparse Row* (CSR) format. As it was described in Sect. 2.5.1, this is a simple and general format for sparse arrays, with minimal storage requirements. The new `CSRShape` class encapsulates 2-dimensional, sparse-matrix domains using the CSR format. It uses two compact arrays to contain the list of existing index elements. The memory space required by this representation is in the order of the number of existing domain elements (or non-zero elements in a sparse matrix).
- **Bitmap** As a second example of sparse domain implementation, the `BitmapShape` class uses a bitmap structure to represent the existing and non-existing indices of a rectangular parallelotope. The memory space required by the bitmap representation is in the order of the parallelotope size, independently of the density of the domain. Although bitmap iterators are less efficient than CSR's to locate the actual elements of the domain, bitmap shapes are more efficient than CSR shapes in terms of memory footprint. Section 4.5.5 of the experimental results contains a more complete comparison of these two formats.

The original `Shape` class included functionalities to expand a shape to generate *ghost borders*, useful in programs with neighbor data synchronization (such as cellular-automata programs). The same functionalities should be defined for sparse domains. The neighborhood property in sparse domains is conceptually different than the one defined for dense domains. We define the neighborhood relationship for sparse domains in terms of graph connectivity. Domain elements are *neighbors* if one of their index coordinates is the same. For a global graph $G = (V, E)$ and a subgraph $G' = (V', E')$ assigned to a local process, so that $V' \in \mathcal{P}(V)$ is a partition of V and $E' \in \mathcal{P}(E)$ is a partition of E. The extended shape of V' is defined by:

$$V'_{ext} = V' \cup \{v \in V,\ \exists(w, v) \in E,\ w \in V'\}$$

Thus, building an expanded shape implies to traverse the local shape once. Using these functionalities, it is possible to program neighbor synchronization codes with graph partitioning, in the same way as for dense matrices.

4.3.2 Tiles

The old `Tile` class is transformed into an implementation of a new `Tile` interface. This interface defines abstract methods to efficiently allocate and retrieve data. It is possible to create different tile implementations for the same kind of shape to use different strategies to store the elements of the domain. For example, a tile implementation can keep the data in a single block of contiguous memory and use index transformations and other ancillary structures to locate the elements in memory. This helps to implement efficient functionalities to traverse and communicate tiles.

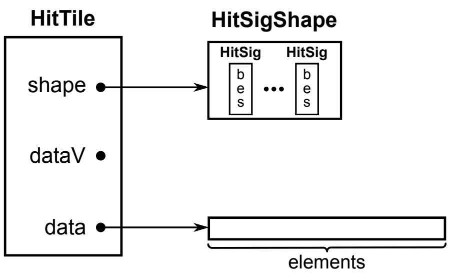
The original version of the Hitmap library was oriented to manage arrays, with only one data element associated to each domain index. The new abstraction also make it possible to create implementations with more than one data space for the same domain. For example, graphs have a single, $N \times N$ square index space, but edges information and vertices values can be stored in different internal data structures (a matrix and a vector, respectively), with different access methods. It is possible to create different tile implementations for matrices, graphs, or other data structures, based on the same shape implementation (see Fig. 4.3).

We have developed new tile implementations for matrices and graphs, with both dense and sparse domains. As it is shown in Fig. 4.4, the different tile implementations reference a shape that defines the domain, and contains one or more pointers to the data sets stored in contiguous memory blocks.

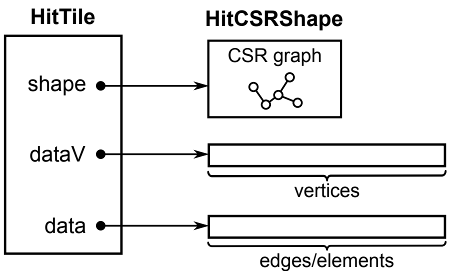
The implementation of the new tile methods to locate data are dependent on the kind of shape implementation. For tiles using the CSR shape, we have chosen a data storage strategy that is directly derived from the CSR format definition. Thus, the tile only stores data for the actual elements. To access a particular element, it is necessary to obtain first its location in the memory using the CSR shape.

For tiles whose domain is defined by a bitmap shape, we have used the same strategy as the tiles that use a shape of signatures. Thus, the tile allocates memory for both existing and non-existing elements. Comparing both same implementations, the tile for bitmaps is almost as efficient as the tile for signatures to retrieve data by using their coordinates. However, there is a small performance penalty due to the extra arithmetic operations involved in the bitmap check before accessing the data.

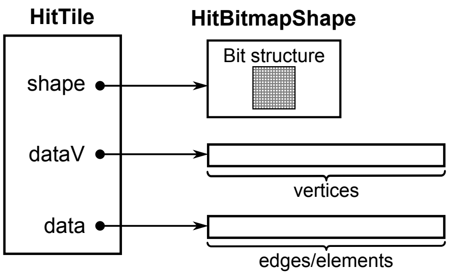
In both CSR and bitmap tiles, retrieving data elements using their indices is not as efficient as with signatures. On the other hand, the iterator methods that traverse the data structure in the proper order, are equally efficient for dense and sparse representations. Thus, the tile iterators are the methods of choice to traverse data in Hitmap, in order to keep better portability when changing from dense to sparse data structures.



(a) Matrix tile using a Signature Shape.



(b) Graph tile using a CSR Shape.



(c) Graph tile using a Bitmap Shape.

Figure 4.4: Internal structure of tile objects with different shape implementations.

4.3.3 Layout

In Hitmap, the layout constructor is a wrapper to call the selected plug-in with an input shape and a topology. The plug-in fills up the fields of the resulting layout object, returning local and/or neighbors subdomain information. The `Shape` class hides the details about how the domain is defined. Nevertheless, each plug-in contains a partition technique which may be appropriate only for a specific kind of domain. For example, regular blocking techniques for signatures are not appropriate for sparse domains, and bisection graph partition techniques are not efficient for dense domains. The `Layout` class does not need relevant changes, but existing layout plug-ins need to be redefined to issue a warning when applied to non-appropriate kinds of shape, and generalized to deal with different kinds of shapes. Internally, the plug-in may select the exact partition technique depending on the shape implementation (see Fig. 4.2).

As an example of partition/mapping techniques appropriate for sparse domains, we have integrated the multilevel k-way partitioning technique found in Metis [80] into a new plug-in. Metis is a state-of-the-art graph partitioning library. Our plug-in translates the information retrieved from the Shape object to the array format expected by the Metis function. After this translation step, the plug-in calls Metis, providing it with the number of virtual processes. Finally, the plug-in uses the results to build the local Shape, and save details to compute neighbor information when requested.

4.3.4 Communication

There is no change in the interface of the communication class. However, the original `Comm` class contained a private method that used the signature shape of the tile to generate an MPI derived datatype. This datatype represented the data location in memory, letting the MPI layer to automatically marshall/unmarshall the data when the communication were invoked. To generalize the communication class for different kinds of shapes, the marshalling functionalities have been moved to the tile interface and tile implementations, and they are called from the communication class when needed.

The new version of the `Comm` object supports two new global communication types to deal with graph information. The first one exchanges neighbor vertices information with other virtual processes, and the second one scatters the data of a whole graph-tile that is stored in one processor. This helps to read graph data from a file in one processor and distribute it across the topology. These new communications are equivalent to the redistribute communications already defined for dense tiles. Other utilities have been added to the library for reading and writing sparse data tiles in different formats, like the Harwell-Boeing format [42], or plain CSR format.

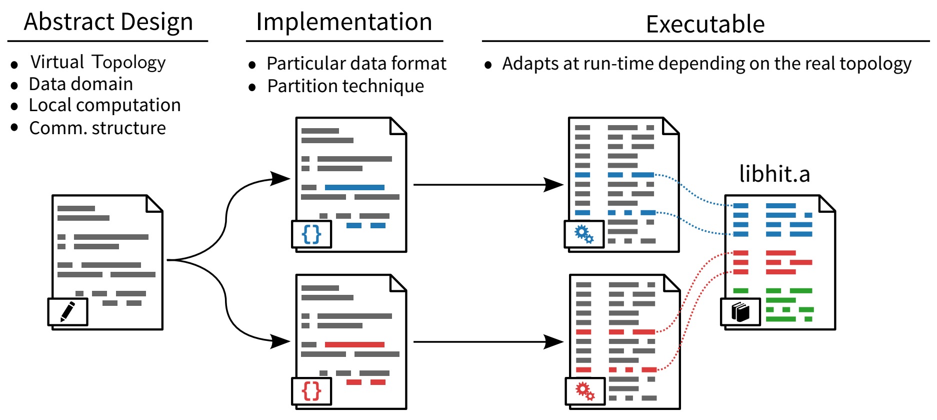


Figure 4.5: Use of the unified interface with the different phases.

4.3.5 Summary

The changes introduced in the Hitmap library allow us to develop applications that use the same abstract design for any type of data structure. Figure 4.5 shows the phases to develop such application. The abstract design contains the decisions about the virtual topology, the data domain, the local computation, and the communication structure. In the implementation phase, the programmer selects the particular data storage format and the partition technique. The resulting executable code adapts at runtime the mapping and communications depending on the current data and the real topology.

4.4 Benchmarks

In this section, we describe three benchmarks developed to evaluate the dense and sparse data integration in the Hitmap library. The first one is a simple sparse matrix-vector multiplication, a benchmark that allows us to compare Hitmap with efficient implementations for sparse data operations, using well-known tools like PETSc [14] (*Portable Extensible Toolkit for Scientific computation*), a library for the parallel solution of systems of equations. The second benchmark performs a simple graph neighbor synchronization. It is used to test the graph partition and distribution support. The last benchmark, which is more complex, calculates the equilibrium position of a 3-dimensional spring system, represented as a graph. It represents a real application and exploits sparse domain functionalities at all levels of the Hitmap programming approach: Partition, layout, and communications. All of them depend on the sparse/dense structure of the input data. We have also implemented this benchmark in PETSc, where many of these features that Hitmap offers have to be implemented manually.

4.4.1 Sparse matrix-vector multiplication benchmark

The first benchmark is a simple sparse matrix-vector multiplication. It is an important kernel in many scientific and engineering applications. It performs the operation $y = Ax$, where the A matrix is sparse and the x and y vectors are dense. The benchmark that we will use is the typical parallelized version of the multiplication, where the compute node that owns element a_{ij} computes the contribution $a_{ij}x_j$ to y_j [136]. As the data partitioning method, we use a one-dimensional data distribution . In this method, each node is assigned matrix elements belonging to a set of consecutive rows. A single matrix-vector product does not have enough computational load to show significant results. Thus, the benchmark performs several iterations using the result as the input for the next iteration ($x_{i+1} \leftarrow y_i$). This requires A to be a square matrix. Moreover, to prevent overflow in the output vector, the matrix elements will be randomly initialized with uniform distribution between 0 and $1/n$, to satisfy $\sum_{j=1}^{n} |A[i][j]| \leq 1\, \forall i$.

Implementations

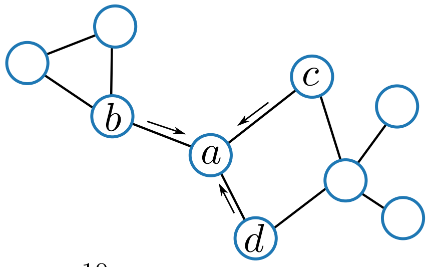
Firstly, we have developed a C version using MPI of the benchmark that uses a CSR sparse matrix. The program includes the code manually developed to perform a row partition of the matrix and to distribute the elements among the processes. Since the result vector is reused in each iteration, all processors need to communicate their contribution to the rest of the processes.

The Hitmap implementation takes the part of the kernel that performs the sequential multiplication of the manual C version. However, it uses the Hitmap data structures to declare the sparse matrix, the layout, and the communication objects to distribute the matrix by rows. Another communication is used to redistribute the result vector in each iteration.

In addition, we have implemented the benchmark using PETSc. We have used the special PETSc matrix and vector objects. These structures are distributed and their elements are assigned internally to processes. The actual multiplication is done using the PETSc matrix-vector multiplication function.

4.4.2 Graph neighbor synchronization benchmark

This benchmark solves a simple problem that involves a computation over a sparse data structure. We extend the idea of neighbor synchronization in FDM (*Finite Difference Method*) applications on dense matrices to civil engineering structural graphs, see e.g. [135]. The application performs several iterations of a graph-update operation. It traverses the graph nodes, updating each node value with the result of a function on the neighbor nodes values. To simulate the load of a real scientific application, we wrote a dummy loop, which issues 10 times the `sin()` mathematical library operation. Figure 4.6 contains an example of a small graph and the mathematical formula the benchmark solves.



$$v_a = \prod_{k=1}^{10} f(\textstyle\sum_i v_i)/n \qquad f() = sin()$$

Figure 4.6: Graph neighbor synchronization benchmark.

C implementation

We have first developed a serial C implementation of the benchmark, to use it as a reference to measure speed-ups and to verify the results of the parallel versions. Our first parallel version includes the code to compute the data distribution manually. It is a highly-tuned and efficient implementation based on the Metis library to calculate the partition of the graph, and MPI to communicate the data between processors. Since the Hitmap library implementation is also based on the same tools, the comparison between performance results between both versions will uncover any potential inefficiency introduced in the design or implementation of Hitmap.

The manual parallel version has the following stages: (1) A sparse graph file is read; (2) the data partition is calculated using the Metis library; (3) each processor initializes the values of the local vertices using the standard ISO C pseudo-random number generator; (4) a loop performs 10 000 iterations of the main computation, including updating the values of each local vertex, and communicating the values for neighbor vertices to other processors; (5) the final result is checked with the help of a hash function.

Hitmap implementation

Using the manual C implementation as a starting point, we have developed a Hitmap version of the program. The Hitmap implementation uses the main computation and other sequential parts of the previous version, adapting them to work with the Hitmap functions for accessing data structures. A new layout plug-in module has been developed to apply the Metis data partition to the Hitmap internal sparse-shape structures. Data-layout and communications have been generated using Hitmap functionalities following the same methodology as for dense structures. In this section, we discuss the Hitmap techniques needed to automatically compute the data-layout, allocate the proper part of the graph, and communicate the neighbor vertices values.

In Listing 4.1, we show the main function of the Hitmap code. Line 2 uses a function to read a graph stored in the file system, and returns a shape object. Then, in Line 5, a virtual topology of processors that uses the internal information available about the real topology is created transparently to the programmer with a single call.

In Line 8, the data-layout is generated with a single Hitmap call. The layout parameters are: (a) The layout plug-in name, (b) the virtual topology of processors generated previously, and (c) the shape with the domain to distribute. The result is a `HitLayout` object, containing the shape assigned to the local processor and information about the neighbors.

In Line 11, we obtain the shape of the local part of the graph with only the local vertices. On the following line, we use the layout to obtain an extended shape with local vertices plus the neighbor vertices from other processors. This is the equivalent to the shape of a tile with a shadow region in a FDM solver for dense matrices. This shape is used to declare and allocate the local tile with double elements (Line 17).

In line 23, a `HitCom` object is created to contain the information needed to issue the communications that will update the neighbor vertices values on each simulation iteration. Data marshaling and unmarshaling is automatized by the communication objects when the communication is invoked (see lines 25 and 33).

Lines 27 to 34 contain the main iteration loop to update local nodes values and reissue communications with a single Hitmap call that reuses the `HitCom` object previously defined.

Listing 4.2 shows the code of the function that updates the local values of the graph. It uses two iterators defined in the library. The first one traverses the local vertices (Line 4), and the second one iterates over the edges to get their neighbor vertices (Line 9). Each neighbor-vertex value contributes to the new value of the local vertex that is set in Line 20.

4.4.3 A FEM benchmark

In this section, we describe a FEM (*Finite Element Method*) benchmark that calculates the equilibrium position of a 3-dimensional spring system, represented as a graph. It represents a real application and exploits sparse domain functionalities at all levels of the Hitmap programming approach: Partition, layout, and communications. The elements are represented by interconnected nodes. There are N free nodes whose positions $\vec{r_i}$ are arranged in a ρ vector, and M fixed nodes. Each node is connected to one or more nodes by a spring. Each spring is assumed to be uniform with the same length l and the same force constant k. Given an initial node configuration, the benchmark will calculate the position where the nodes are in equilibrium under forces applied to them.

Figure 4.7 shows a 2-D example of a simple spring system. The first part of the figure contains the initial node position and the second one contains the final equilibrium position after the benchmark execution.

```c
// Read the global graph.
HitShape shape_global = hit_shapeHBRead("graph_file.rb");

// Create the topologoy object.
HitTopology topo = hit_topology(plug_topPlain);

// Distribute the graph among the processors.
HitLayout lay = hit_layout(plug_layMetis,topo,&shape_global);

// Get the shapes for the local and local extended graphs.
HitShape local_shape = hit_layShape(lay);
HitShape ext_shape = hit_layExtendedShape(lay);

// Allocate memory for the graph.
size_t sdouble = sizeof(double);
HitTile_double graph;
hit_tileDomainShapeAlloc(&graph,sdouble,HIT_NONHIERARCHICAL,ext_shape);

// Init the local graph.
init_graph(graph, shape_global);

// Create a communicator to send the values of the neighbor vertices.
HitCom com = hit_comSparseUpdate(lay, &graph, HIT_DOUBLE);
// Send initial values
hit_comDo(&com);

int i;
// Update loop.
for(i=0; i<ITERATIONS; i++){
    // Update the graph.
    synchronization_iteration(local_shape,ext_shape,&graph);
    // Communication.
    hit_comDo(&com);
}
```

Listing 4.1: Hitmap main function of the graph neighbor synchronization benchmark.

```c
int vertex, edge;

// Iterate through all the vertices.
hit_sparseShapeVertexIterator(vertex,local_shape){

    // Set new value to 0.
    double value = 0;

    hit_sparseShapeEdgeIterator(edge,ext_shape,vertex){

        // Get the neighbor.
        int neighbor = hit_sparseShapeEdgeTarget(ext_shape,edge);
        // Add its contribution.
        value +=  hit_tileElemAt(1,graph,neighbor);
    }

    // Dummy workload = 10.
    int i;
    for(i=0; i<WORKLOAD; i++){
        value = sin(value+1);
    }

    int nedge = hit_sparseShapeNumberEdgesFromVertix(ext_shape,vertex);

    // Update the value of the vertex.
    hit_tileElemAt(1,graph_aux,vertex) =  (value / nedge);
}
```

Listing 4.2: Function that serially updates the local part of the graph in the Hitmap version of the graph neighbor synchronization benchmark.

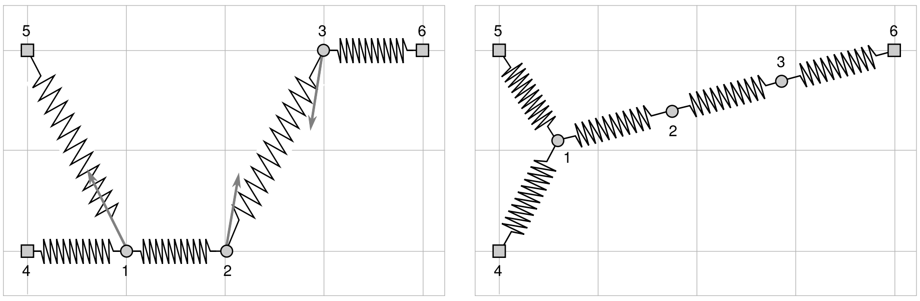


Figure 4.7: Spring system example: Initial node position and the resulting forces (left) and final equilibrium position (right).

Finite element methods in a nutshell

The finite element method (FEM) is a computational technique used to obtain approximate solutions of boundary value problems in engineering [73]. A boundary value problem is a mathematical problem in which one or more dependent variables must satisfy a differential equation within a known domain, together with a set of additional constraints on the boundary of the domain.

The domains used in FEM methods often represents a physical structure. Depending on the problem being analyzed, the variables could be, for example, physical displacement, heat transfer, or fluid dynamics. The solution of these kind of physical processes can not be obtained by using exact analytical methods, due to its complexity. The FEM method reduces the complexity of the problem by performing a discretization, using simple connected parts called finite elements. Many simple equations are solved over the finite elements to approximate the more complex equation over the complete domain.

The process of dividing the domain into elements is known as *discretization*. Figure 4.8 shows an example of a bar that is represented using several, similar elements. A classic method would calculate the displacement along the bar due to the force using differential equations. Instead of that, a FEM method solves simple equations for each finite element of the bar. The points located at the corners of the elements are called *nodes* and the set of all elements within a domain is called a *mesh*. The approximation function used to solve the problem describes how the unknown variable (e.g. displacement) changes at each node. The *assembly process* consists in combining together the elements to provide an approximate solution for the entire domain.

FEM Challenges

The parallelization of a FEM algorithm present many challenges. The FEM methods model structures as series of small connected parts that are usually represented by a graph structure.

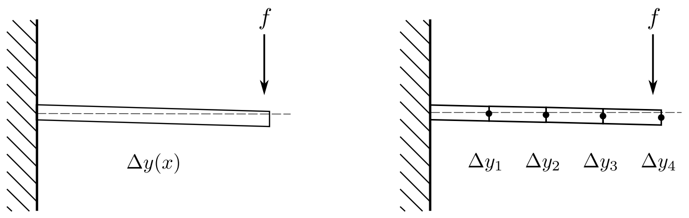


Figure 4.8: A bar with a force applied and its discretization using FEM.

Thus, the parallel version has to deal with the manipulation, distribution, and communication of sparse data. Moreover, they present great load unbalances because the sparse structures are usually irregular. Thus, the operations involved in calculate one finite element vary depending on the current domain part being processed.

Mathematical background

We briefly show the mathematical background of the FEM benchmark. To obtain the equations that determine the equilibrium position, we have to start from the potential energy. In this system, the potential energy is composed by the strain energy of elastic distortion in each spring. The following equation calculates the potential energy. The parameters are the current positions of the free nodes.

$$V(\vec{\rho}) = \frac{1}{2}\sum_{i<j} k(l - \|\vec{r}_i - \vec{r}_j\|)^2 \qquad \begin{matrix} i\in[1,N] \\ j\in[1,N+M] \end{matrix}$$

Equilibrium position corresponds with the configurations that make the gradient of the potential energy function equal to zero, that is: $\nabla V(\vec{\rho}) = 0$.

We have selected the Newton iterative method to find the root in the previous equation. The Newton method, also called the Newton-Raphson method, requires the evaluation at arbitrary points of both the function and its derivative. The Newton method geometrically extends the tangent line at a current point x_i until it crosses zero, and then sets the next guess x_{i+1} to the abscissa of that zero-crossing. Its great advantage is that it obtains the solution faster than other methods because it has a quadratic rate of convergence [105]. The generalized version of the Newton-Raphson method for multiple dimensions needs the derivative of the gradient function, that is, the Hessian matrix of the potential energy function: $V(\vec{\rho})$. A Hessian matrix is a square matrix composed by the second partial derivatives of the function. The iterative method uses successive approximations to obtain a more accurate solution. In each step, it calculates

$$\vec{\rho}_{k+1} = \vec{\rho}_k - \vec{\xi}_k$$

where $\vec{\xi}_k$ is

$$\text{Hess}\, V(\vec{\rho}_k) \cdot \vec{\xi}_k = \nabla V(\vec{\rho}_k)$$

The previous equation is a linear system that can be rewritten in the $Ax = b$ form, where the Hessian matrix function evaluated at the current position is the coefficient matrix, the vector $\vec{\xi}_k$ has the unknown variables, and the gradient function evaluated at the current position is the constant vector. To solve this system we have selected the Jacobi iterative method [56].

Parallel algorithm for the FEM benchmark

The parallel algorithm of the FEM benchmark divides the system into irregular pieces, and executes several computation stages which need neighbor synchronization. It performs the following stages:

1. Graph load: The process designated as master loads the sparse graph representing the structure of the spring system, and the position coordinates of the nodes from the file-system.
2. Node initialization: The master process sets randomly the 10% of the nodes as fixed. The coordinates of the remaining nodes (free nodes) are the variables of the benchmark.
3. Graph partition: The data partition is calculated by the master.
4. Graph distribution: The master process sends the nodes coordinates and their *free/fixed* status to the appropriate processor.
5. Approximation computation: Every process performs a loop with N iterations of the Newton method to get an approximation of the equilibrium position. At each iteration, the current gradient and Hessian matrix is calculated to generate a linear system which solution determines the new approximation. This stage includes the communication of the new approximation coordinates. This communication is a neighbor synchronization where the processes update the value at the ghost border.
6. System solution: For each iteration of the main loop, the Jacobi method performs J iterations to solve the linear system. This stage includes the communication of the intermediate solutions of neighbor nodes across processes on each iteration.
7. Result check: All the processes do a reduction to obtain the norm of the gradient vector and to verify that the benchmark has reached the equilibrium position.

```
int i, j, vertex;
// Load the graph from the file.
HitGraphTileCSR global_graph = init_graph();
// Create the topology object.
HitTopology topo = hit_topology(plug_topPlain);
// Distribute the graph among the processors.
HitLayout lay = hit_layout(plug_layMetis, topo, hit_tileShape(global_graph));
// Get the local shape and the extended shape.
HitShapeCSR local_shape = hit_layShape(lay);
HitShapeCSR ext_shape = hit_cShapeExpand(local_shape,hit_tileShape(global_graph),1);
// Allocate memory of all the variables.
HitGraphTileCSR local_graph, e;
hit_gcTileDomainShapeAlloc(&local_graph, Node, ext_shape, HIT_VERTICES);
hit_gcTileDomainShapeAlloc(&e, Node, ext_shape, HIT_VERTICES);
// Create the communication objects.
HitCom comA = hit_comSparseScatter(lay, &global_graph, &local_graph, HitNode);
HitCom comB = hit_comSparseUpdate(lay, &local_graph, HitNode);
HitCom comC = hit_comSparseUpdate(lay, &e, HitVector);
// Send all the data from the root proc to the other procs.
hit_comDo( &comA );
hit_comDo( &comB );
// Main loop for the newton method
for(i=0; i<ITER1; i++){
   // Iterate trough all the vertices and
   // obtain the gradient and the hessian.
   hit_cShapeVertexIterator(vertex, local_shape){
      Node current = hit_gcTileVertexAt(local_graph, vertex);
      if (!current.fixed) calculate_GH(vertex);
   }
   // Loop for the jacobi method to solve the system.
   for(j=0; j<ITER2; j++){
      // Perform a iteration.
      hit_cShapeVertexIterator(vertex, local_shape){
         Node current = hit_gcTileVertexAt(local_graph, vertex);
         if (!current.fixed) solve_system_iter(vertex);
      }
      // Update the displacement.
      hit_gcTileCopyVertices(&e, &new_e);
      hit_comDo( &comC );
   }
   // Update the position with the final displacement.
   hit_cShapeVertexIterator(vertex, local_shape){
      Node * current = & hit_gcTileVertexAt(local_graph, vertex);
      if (!current->fixed)
         subV(current->r, current->r, hit_gcTileVertexAt(e, vertex));
   }
   hit_comDo(&comB);
}
```

Listing 4.3: Main function of the Hitmap implementation of the FEM benchmark.

Hitmap implementation of FEM benchmark

In this section, we discuss how to implement the FEM parallel algorithm using Hitmap, paying special attention to the functionalities to automatically compute the data layout, and communicate the neighbor's vertices values.

Listing 4.3 shows the main function that contains the complete parallel code of our FEM algorithm implementation. Line 3 calls a function that loads the global graph information and coordinates from files. Line 5 transparently generates a virtual topology of processors using the internal information available about the real topology. We have selected a plain topology, that does not define specific neighborhood relationships. Lines 7 to 12 correspond to the graph domain partition. In Line 7, the data-layout is generated with a Hitmap call. The layout parameters are: (a) The layout plug-in name; (b) the virtual topology of processors generated previously; and (c) the shape with the domain to distribute. The plug-in internally calls the Metis function to determine the distribution of vertices. The result is a `HitLayout` object. In Line 9, we obtain the shape of the local part of the graph containing just the local nodes. On the following line, we use the shape returned by the layout to obtain an extended shape with local nodes plus neighbor nodes located at other processors. The parameters of the function are: The current local shape, the global shape, and the distance of the nodes to add. Line 12 declares and allocates the local extended graph.

From Lines 16 to 18, three HitCom objects are created to be used in the following stages. The first object (comA) is used to perform the communication that will distribute the original node values to the processor where the data has been mapped. The second object (comB) contains the information for the communication that will update the neighbor nodes values in the Newton method. The last object (comC) is created for the communication that will update the neighbor approximation values in the Jacobi method. Lines 20 and 21 invoke the communications to distribute the graph to the processes. The first one, distributes the data from the master process to the appropriate process as defined by the graph partition. The second one is used to update the ghost border.

The main loop of the Newton method starts at Line 23. This loop uses an iterator defined in the library to traverse the local vertices of the graph (Lines 26 to 29), calculating the gradient and Hessian for the Newton method with the sequential function `calculate_GH`. The inner loop for the Jacobi method starts at Line 31. This loop traverses the free nodes to get an approximation for the linear system. Then, the new solution of the linear system is copied from the tile `new_e` to the tile `e` (Line 38) and the Line 39 invokes the communication to update the neighbor values of this tile that are needed by other processors on the next iteration. At this point, the inner loop for the Jacobi method is over. Line 42 uses an iterator to update the positions of the nodes. Finally, Line 47 updates the neighbor nodes position.

4.5 Experimental results

Experimental work has been conducted to show that: (1) The new version of Hitmap can be used to program dense and sparse application with the same methodology; (2) the abstractions introduced by the library simplify the complexity of codes that deal with the sparse domains manually, without introducing significant performance penalties.

For each introduced benchmark, we have developed three program versions. The first one codifies an *ad-hoc* implementation of the CSR format to represent a sparse matrix or a graph. It either manually deals with the calls to Metis partitioning system or implements its own partition method (as in the case of the multiplication benchmark). In this version, the communications are done using MPI calls. The second one is an equivalent program written with the Hitmap library. For the multiplication and FEM benchmarks, we also consider a third implementation with the PETSc library, using sparse PETSc matrices representation, and the solvers for linear systems included in this library. Finally, for the FEM experiment we compare two additional versions: Hitmap using a row partition instead of Metis and PETSc using its dense matrix format.

The Hitmap kernel code is minimum dependent upon the tile subclass used. Therefore, different extensions of the tile class can be used just by choosing a different name. This feature allowed us to use the same code to experiment with all of the three shape implementations, in order to compare their efficiency for different graph densities.

Listing 4.4 shows an example with two codes with the same program, one for dense and other for sparse structures, where the changes are highlighted. All processes declare the shapes of the whole data structures. Then, the virtual topology is selected. After that, the partition and mapping procedure is activated. Once mapped, and after the memory allocation, we can use data in the local subdomain.

Since Hitmap is implemented using C, which is a functional language, some access functions are in fact dependent on the tile class. We are working in a Hitmap prototype implemented in C++ that uses polymorphism to complete decouple the access code from the type of tile. With this Hitmap version, it would be possible to choose the tile type without affecting the rest of the code.

4.5.1 Experiments design

We use three different experimental platforms with different architectures: A multicore shared-memory machine, and two distributed clusters of commodity PCs:

- **Geopar:** The shared-memory system, Geopar, is an Intel S7000FC4URE server with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32Gb of RAM.

- **Beowulf DC:** The first distributed system is a homogeneous Beowulf cluster of up to 20 Intel Core2 Duo nodes at 2.20GHz and 1Gb of RAM each,

```c
// Load the global matrix.
HitShape sglobal = hit_shapeStd(2,ROWS,COLS);

// Create the topology object.
HitTopology topo = hit_topology(plug_topArray2D);

// Distribute the matrix among the processors.
HitLayout lay = hit_layout(layBlocks,topo,&sglobal);

// Get the shape for the local matrix.
HitShape shape = hit_layShape(lay);

// Allocate the matrix.
HitTile_double M;
hit_tileDomainShapeAlloc(&M, double, shape);

// Init the matrix values.
int i,j;
hit_shapeIterator(j,shape,0){
   hit_shapeIterator(j,shape,1){
      hit_tileElemAt(2,M,i,j) = 0.0;
   }
}
```

```c
// Load the global matrix.
HitShape sglobal = hit_fileHBMatrixRead("file.rb");

// Create the topology object.
HitTopology topo = hit_topology(plug_topPlain);

// Distribute the matrix among the processors.
HitLayout lay = hit_layout(laySparse,topo,&sglobal);

// Get the shape for the local matrix.
HitShape shape = hit_layShape(lay);

// Allocate the matrix.
HitTile_double M;
hit_mcTileDomainShapeAlloc(&M, double, shape);

// Init the matrix values.
int i,j;
hit_cShapeRowIterator(i,shape){
   hit_cShapeColumnIterator(j,shape,i){
      hit_mcTileElemIteratorAt(M,i,j) = 0.0;
   }
}
```

Listing 4.4: Dense and sparse hitmap management example with highlighted changes.

- **Beowulf SC:** The second one is another Beowulf cluster, composed by 19 AMD Athlon 3000+ single-core processors at 1.8GHz and 1Gb of RAM each.

Both clusters are interconnected by 100Mbit Ethernet networks. The compiler used is GCC version 4.4. The benchmarks codes, Hitmap library, and PETSc v3.2 library have been compiled with -O3 optimization. The MPI implementation used is MPICH2 v1.4.

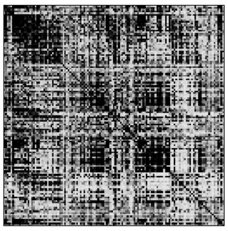
In order to expose the effects of exploiting multi-core processors in a cluster architecture, we have run the experiments in both clusters using up to twice as many processes as physical nodes available. In the Beowulf DC cluster each process is executed by a different core. But, in the Beowulf SC cluster, the mono-core processors execute more than one process when there are more processes than nodes.

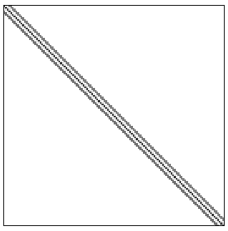
We have conducted experiments with sparse matrices and graphs with different characteristics. We define the *density degree* (d) as the number of edges divided by the square of the number of vertices: $d = |E|/|V|^2$. We use some input examples from the Public Sparse Matrix Collection of the University of Florida [38] representing typical graphs modeling real 3-dimensional structures, with very low density degree.

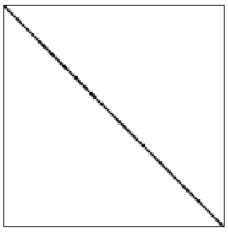
Table 4.2 shows the properties of the matrices used as examples while Fig. 4.9 shows a representation of the location of non-zero elements in the matrices. The first four ones are used for the matrix-vector multiplication benchmark. They have a similar high number of non-zero elements to create a big load. However, they present a large difference in the number of rows and densities, deriving in very different communication sizes. The next two ones are used in the graph synchronization benchmark. And the last two ones are used for the FEM benchmark. In this case the number of rows is similar to produce similar partition sizes but we select examples with very different number of edges (non-zero elements) to produce very different computational and communication load in this benchmark.

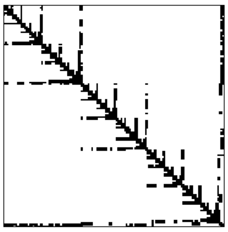
We also experiment with random graphs generated using the PreZER implementation of the Erdős-Rényi $\Gamma_{v,p}$ model [96]. This model chooses a graph uniformly at random from the set of graphs with v vertices where each edge has the same independent probability p to exist. This method allow us to build graphs with higher degrees of density that may represent certain relationships such as those found in social networks or economy transactions. We use this inputsets for the FEM benchmark to observe the behavior of the different implementations when the density changes.

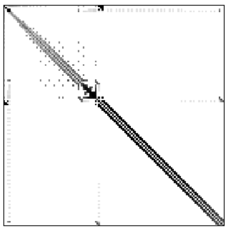
The matrix-vector multiplication benchmarks have been run using 100 iterations of the algorithm explained in Sect. 4.4.1. The graph synchronization benchmark have been run using 10 000 iterations. In each iteration the sin function is called 10 times. For the spring benchmark we have fixed the number of iterations instead of using a convergence condition check. because some graphs in the input set do not meet the convergence conditions of the Jacobi method. The FEM benchmark has been run using 20 iterations for the Newton method, and 100 iterations for the Jacobi method. We have also executed the FEM benchmark using, randomly generated graphs of 1 000 vertices with density degrees from 0.1 to 0.9.

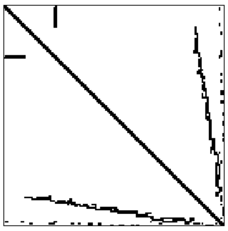

(a) human_gene2

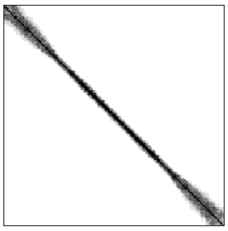

(b) atmosmodm


(c) af_shell1


(d) pkustk14


(e) bodyy6


(f) pwt


(g) filter3D

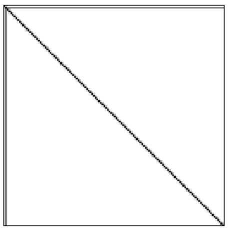

(h) lung2

Figure 4.9: Non-zero structure of the input example matrices.

Matrix/Graph	Rows/Nodes	Non-zero	d
(a) human_gene2	14,340	18,068,388	8.8 $\times 10^{-2}$
(b) atmosmodm	1,489,752	10,319,760	4.6 $\times 10^{-6}$
(c) af_shell1	504,855	17,562,051	6.9 $\times 10^{-5}$
(d) pkustk14	151,926	14,836,504	6.4 $\times 10^{-4}$
(e) bodyy6	19,366	134,208	3.6 $\times 10^{-4}$
(f) pwt	36,519	326,107	2.4 $\times 10^{-4}$
(g) filter3D	106,437	2,707,179	2.4 $\times 10^{-4}$
(h) lung2	109,460	492,564	4.1 $\times 10^{-5}$

Table 4.2: Characteristics of some input set examples for: (a)-(d) vector-matrix multiplication, (e)-(f) graph synchronization and (g)-(h) FEM benchmark.

4.5.2 Performance

Figures 4.10 and 4.10 show the execution times (without initialization times) obtained for the matrix-vector multiplication benchmark in the three architectures. The vertical line that appears on the clusters plots marks when there is more than one mapped process per node. In the cluster architectures (SC Beowulf and DC Beowulf), the times for all versions with one process are the same. Meaning the sequential parts of the applications behave similarly, the differences are in the parallel parts. PETSc obtains slightly better performance for small number of processors, but the scalability trend is the same for all versions when the maximum number of processors approximates to the number of nodes. In the shared-memory architecture (Geopar) we observe the same results in the three versions for some example matrices (e.g. human_gene2 or atmosmodm). For other example matrices (e.g. af_shell1 or pkustk) the PETSc version leads to half the execution time in the sequential part of the code. This difference appear to be related with the PETSc storage policy and/or internal optimizations. It is not possible to determine the exact reason without a deeper analysis of the PETSc internals, as this effect appears for matrices with different structures and densities, and it does not appear in other similar ones (see Table 4.2 and Fig. 4.9). As the number of processes increases the communication times are relatively higher and the effect of the sequential optimizations have less impact.

Figure 4.12 shows the speed-up for both manual C and Hitmap benchmark implementations. There is no significant difference between the two implementations in terms of performance. Therefore, the abstractions introduced by Hitmap (such as the common interface for dense and sparse data structures, or the adaptation of the partition technique in the plug-in module system), do not lead to performance reduction comparing with the manual version.

Figure 4.13 shows the execution times of the FEM benchmark, in the three considered architectures. The manual C and the Hitmap versions present some irregularities for some

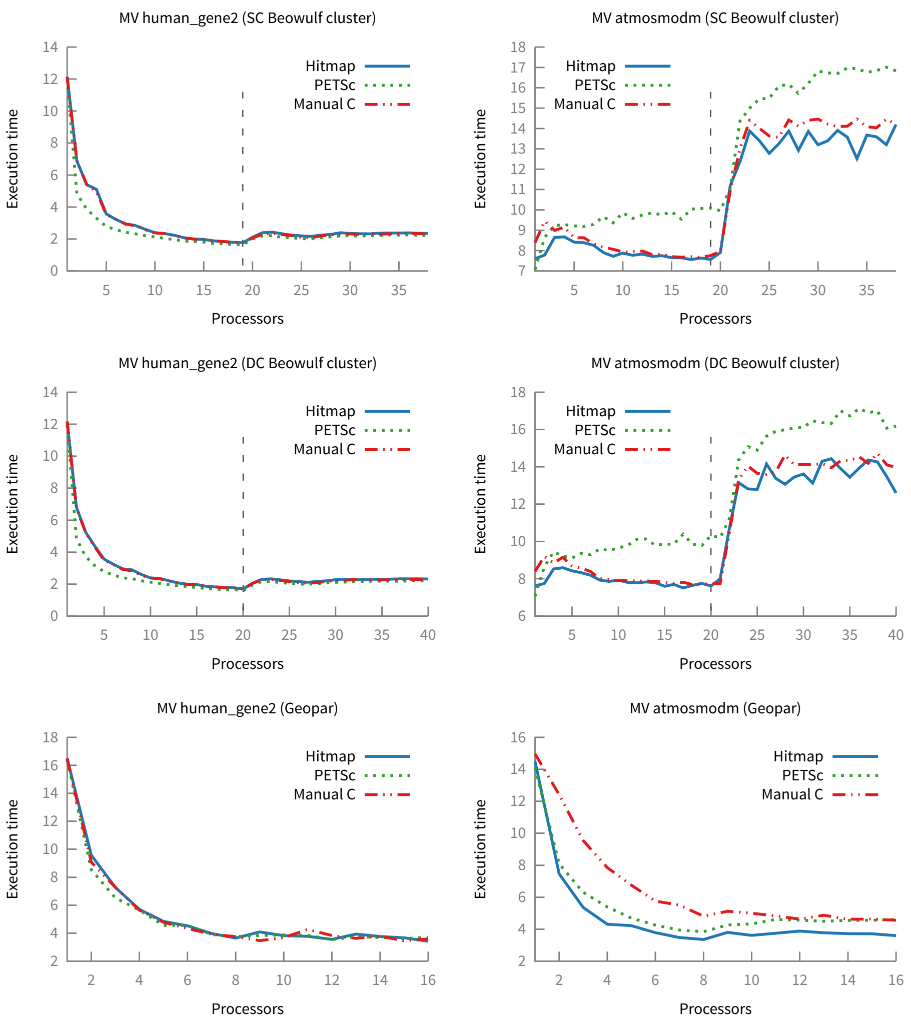


Figure 4.10: Execution time comparison for Hitmap, manually developed, and PETSc implementations of the MV (Matrix-Vector multiplication) benchmark using the (a) and (b) matrix examples.

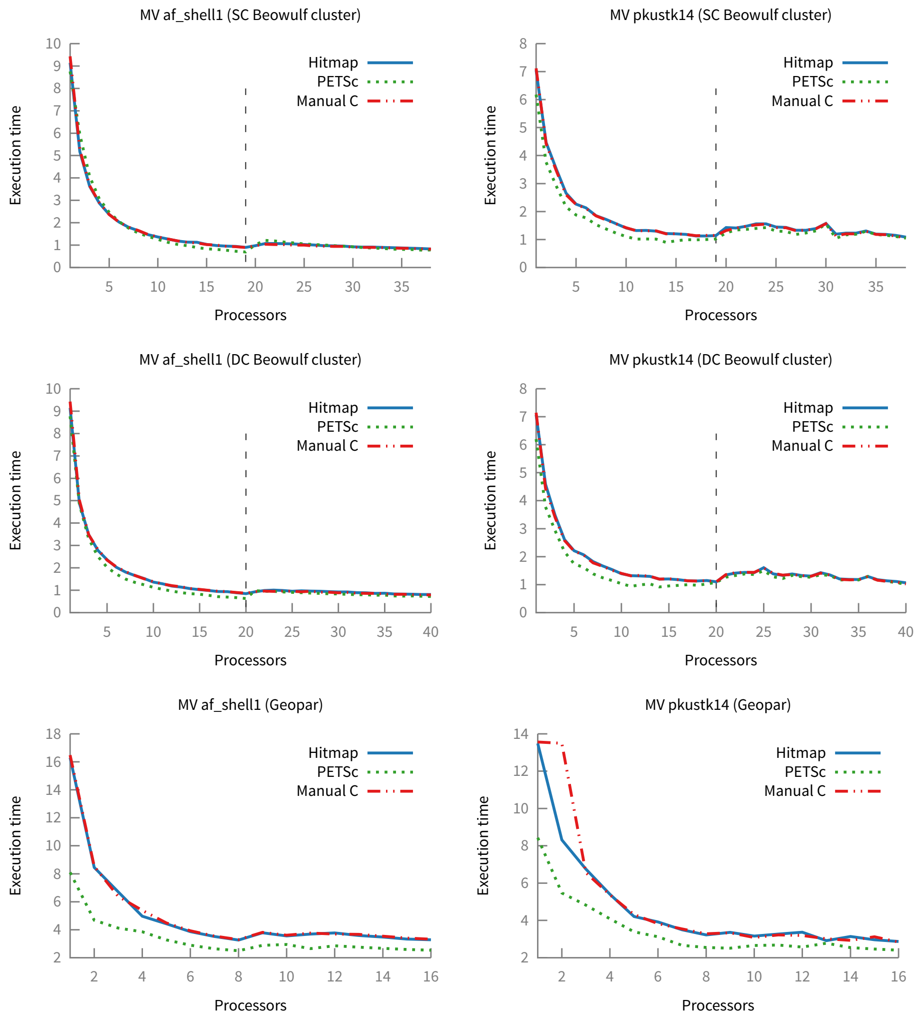


Figure 4.11: Execution time comparison for Hitmap, manually developed, and PETSc implementations of the MV (Matrix-Vector multiplication) benchmark using the (c) and (d) matrix examples.

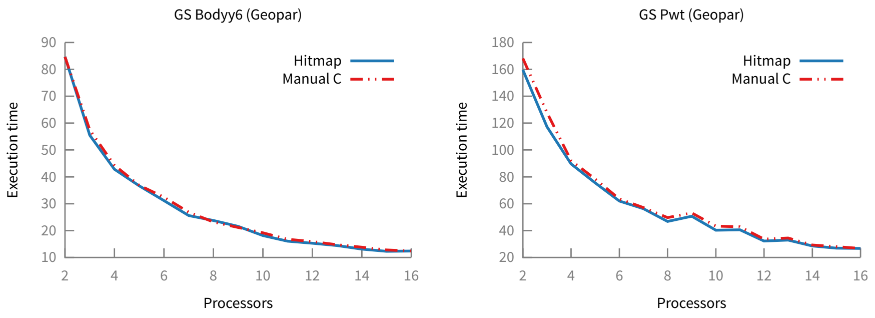


Figure 4.12: Execution time comparison for Hitmap and manually developed implementations of the GS (Graph Synchronization) benchmark using the (e) and (f) graph examples in Geopar.

specific numbers of processes. This is mainly derived from the results of the Metis partitioning policy used. Typically it gets advantage of the graph structure to obtain a well suited and balanced distribution. For the simple graph structure of the example, sometimes the technique selected by default does not minimize the number of edges across different parts, deriving in more expensive communications than a simpler row-blocks partition like the one used by PETSc. In the shared-memory architecture where communications are faster these irregularities have much less impact. Despite these irregularities the behavior is similar as the one observed for the previous benchmark. The sequential part of the computation is faster for PETSc, hiding the advantages of the Hitmap version until a significant number of processors is selected. We may also observe that for the case of oversubscription (launching more processes than processing elements available) the system with more processes than nodes, the performance of PETSc degrades while Hitmap maintains the behavior of the manual version.

Finally, Figs. 4.14 and 4.15 show the execution times obtained with the FEM benchmark for different input graph densities. For this experiment set we also compare Hitmap using a row partition and PETSc using its dense matrix format. The figures show the results in the single-core cluster for 1, 2 and 10 processors. We have selected these experiments to show the sequential performance and the parallel behavior with a small and a bigger number of processes. For the sequential execution, all versions have a similar behavior except the PETSc dense version that, as expected, has the same performance regardless the density. The Hitmap versions show the same result as the manually developed version and they have better performance than the PETSc sparse version. Hitmap iterators are more efficient than PETSc data accesses for degree densities that are not very low. In addition, the communications in Hitmap are also more efficient. Thus, the relative performance between Hitmap and PETSc improves when the number of processors grows.

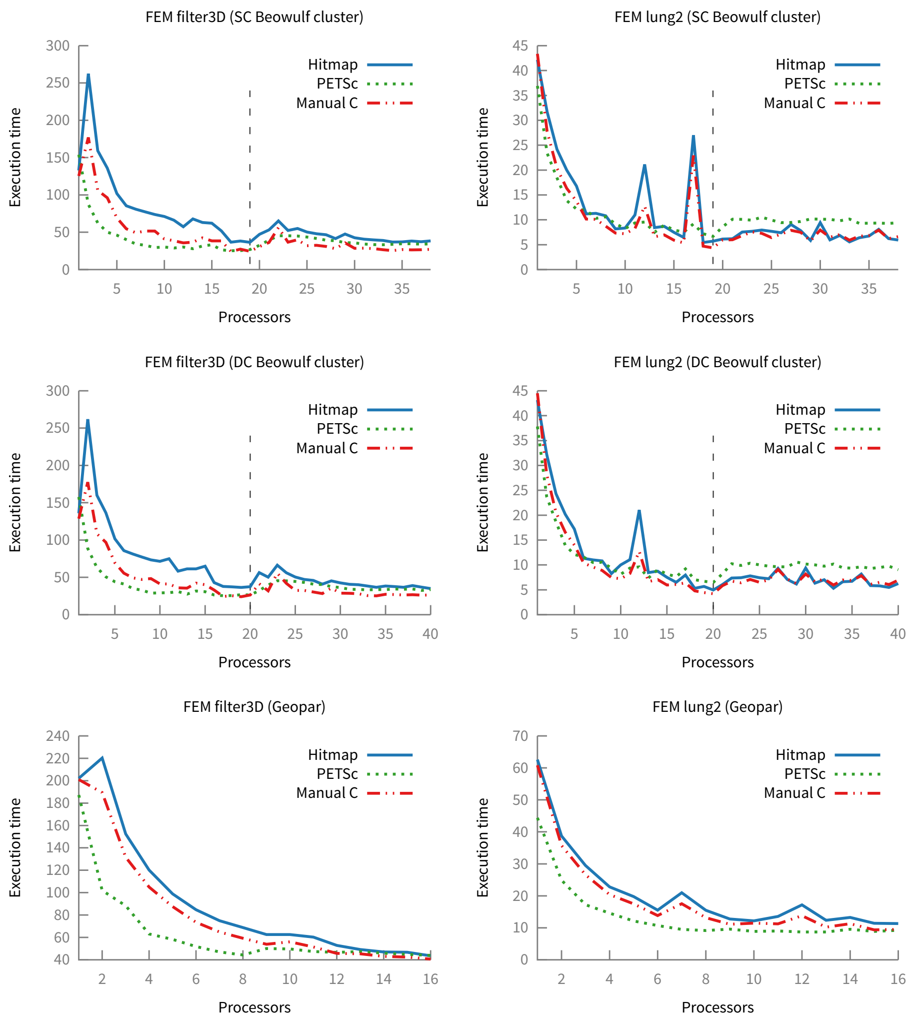


Figure 4.13: Speed-up comparison for Hitmap, manually developed, and PETSc implementations of the FEM (Finite Element Method) benchmark using the (g) and (h) matrix examples.

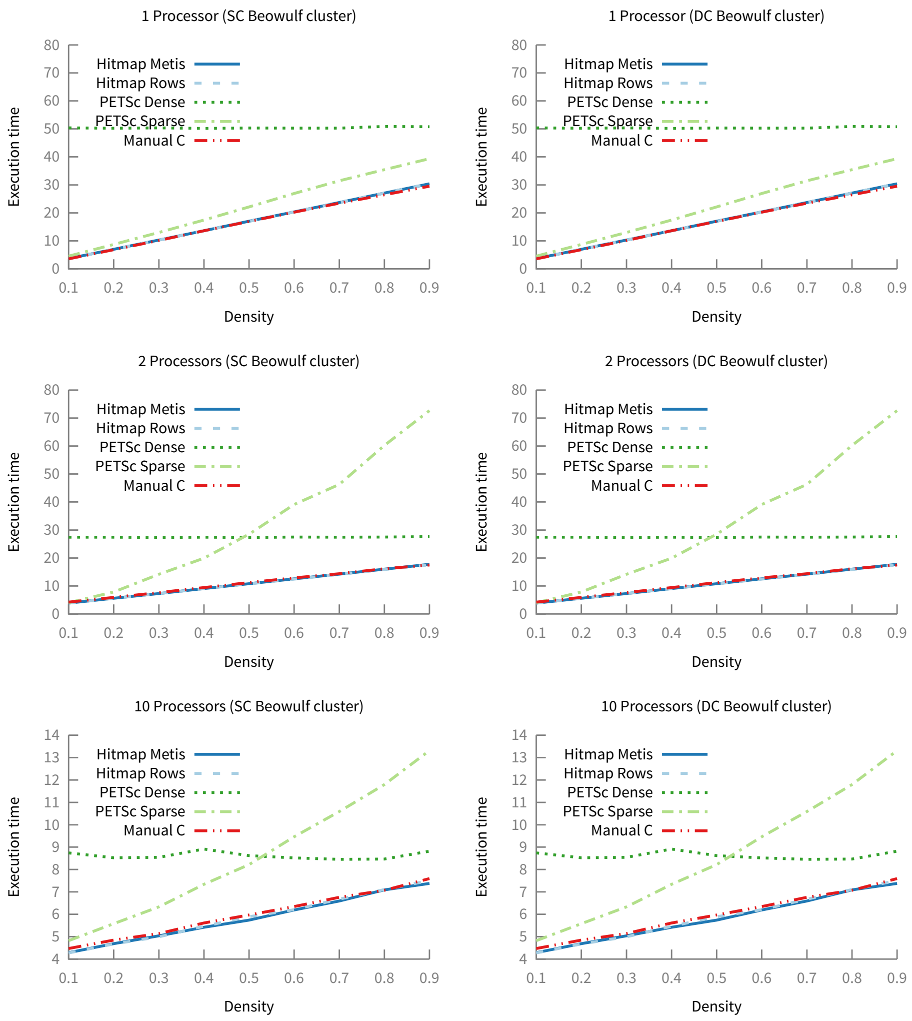


Figure 4.14: Execution time comparison for Hitmap (Metis and Row partition), manually developed, and PETSc (Dense and Sparse data structures) implementations of the FEM (Finite Element Method) benchmark using random generated matrices with different densities in the single and dual core clusters.

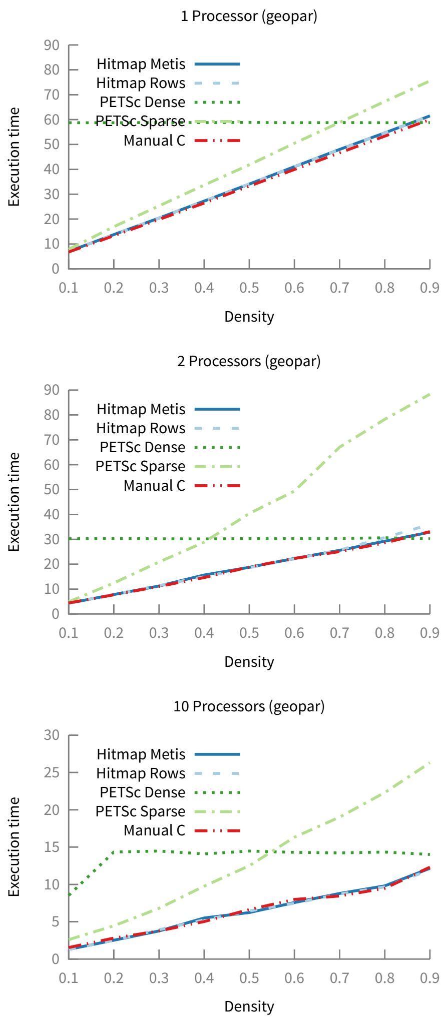


Figure 4.15: Execution time comparison for Hitmap (Metis and Row partition), manually developed, and PETSc (Dense and Sparse data structures) implementations of the FEM (Finite Element Method) benchmark using random generated matrices with different densities in the geopar server.

4.5.3 Code complexity

To compare Hitmap code complexity with respect to the manually-parallelized and the PETSc implementations, we present several code complexity and development-effort metrics. Figure 4.16 compares the number of code lines and the number of tokens. We distinguish lines devoted to sequential computation, declarations, parallelism (data layouts and communications), internal code of the solvers, and non-essential lines (input-output, timers, etc). In the case of PETSc, the solvers codes are internal library functions called directly at the top level. Thus, the are not included in the analysis. For the other versions, we have also taken into account the solvers because they are part of the application code. Internal functions of Hitmap and PETSc devoted to implement data partitions, data accesses, marshaling, communications, etc, are skipped in this analysis.

For the matrix-vector multiplication benchmark, Hitmap reduces the total number of code lines in 56% comparing with the manual MPI implementation and 37% comparing with PETSc. It is remarkable that lines specifically devoted to parallelism control are reduced in 87% with respect to the manual version. Taking into account the multiplication solver that is part of PETSc and that it has to be programmed for the other versions, PETSc has only a reduction of 30%.

For the graph synchronization benchmark, and taking into account only essential lines, the use of Hitmap library leads to a 72% reduction on the total number of code lines with respect to C version. Regarding lines devoted specifically to parallelism, the percentage of reduction is 79%.

For the FEM benchmark, Hitmap also achieves a great reduction of the number of lines: 37% comparing with the manual implementation and 61% comparing with PETSc. Lines specifically devoted to parallelism control are reduced in 88% compared with the manual version. The PETSc version needs a higher number of lines to program the Jacobi solver within the library.

Tables 4.3, 4.4 and 4.5 show the McCabe's cyclomatic complexity metric for each function of the three cases of study. As can be seen, the total cyclomatic complexity of Hitmap is less than half the value of the manual version for all benchmarks. The reason for the reduction of complexity in the Hitmap versions is that Hitmap greatly simplifies the programmer effort for data distribution and communication, compared with the equivalent code to manually calculate the information needed by the MPI routines. PETSc also shows a good complexity reduction for the matrix-vector multiplication benchmark. However, it increases the complexity for the FEM benchmark. As PETSc does not have a complete support for graph operations, it needs similar code than the Manual C version to generate the Hessian matrix from the elements positions, to feed the solver on each iteration. The complexity increase is due to the implementation of the functions for the linear solvers and matrix operations.

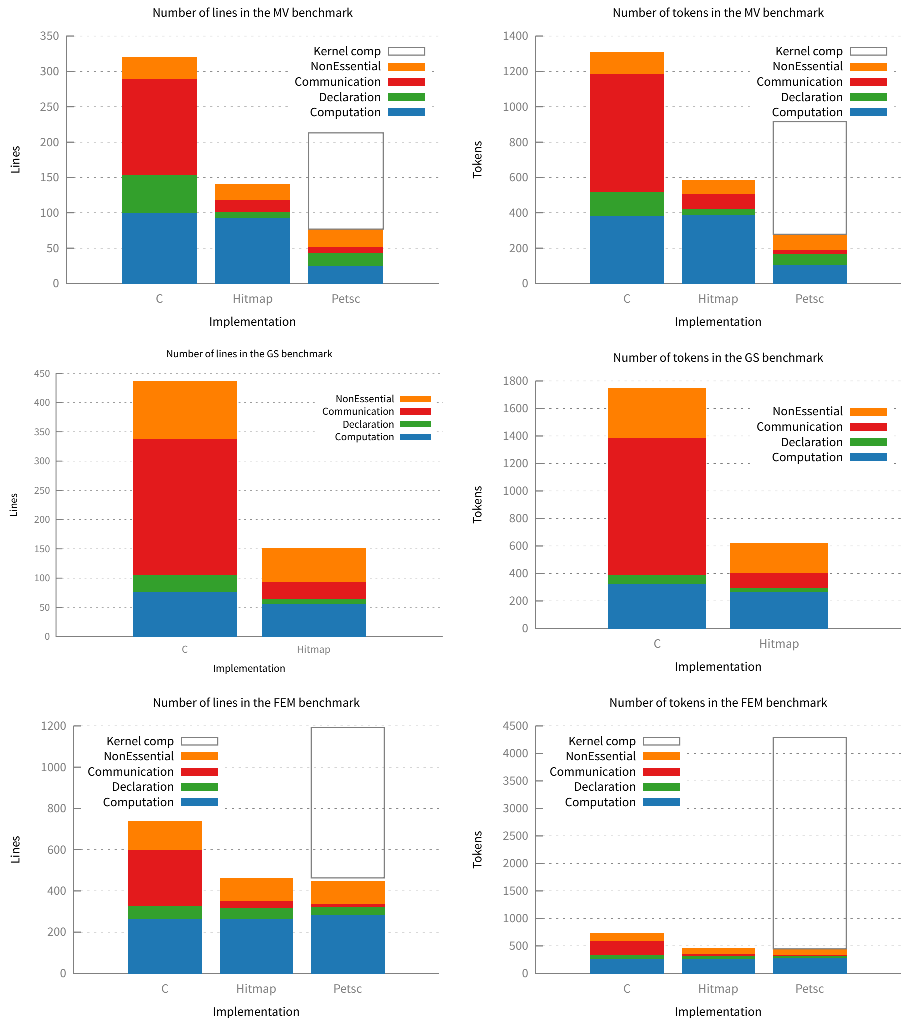


Figure 4.16: Comparison of number of code lines and number of tokens.

Function	C	Hitmap	PETSc
Main	6	5	2
Matrix and vector initialization	8	6	3
Matrix and vector partition	15	-	-
Multiplication	3	1	-
Mult solver	-	-	10
Mult solver parallel	-	-	2
Mult solver computation	-	-	11
Vector redistribution	12	-	2
Vector norm	2	3	-
Other (Frees, Timers, etc)	4	2	-
Total	50	17	28

Table 4.3: Cyclomatic complexity of the multiplication benchmark.

Function	C	Hitmap
Main	5	4
Read graph	2	-
Graph partition	39	-
Init graph	6	6
Iteration	4	2
Graph alltoall	3	-
Result check	2	2
Other	9	2
Total	70	16

Table 4.4: Cyclomatic complexity of the graph synchronization benchmark.

Function	C	Hitmap	PETSc
Main	11	7	2
Init graph	6	5	6
Init structures	5	4	-
Read coordinates	11	12	14
Preallocate system matrix	-	-	8
Graph partition	44	-	-
Calculate system	6	5	12
Solve system	5	4	-
KSP solver	-	-	88
Jacobi solver	-	-	6
Mult solver	-	-	10
Mult solver parallel	-	-	2
Mult solver computation	-	-	25
Vector mult solver	-	-	13
Gradient norm	5	4	-
Scatter graph	5	-	-
All to all graph	6	-	-
Other	6	10	3
Total	110	51	189

Table 4.5: Cyclomatic complexity of the FEM benchmark.

4.5.4 Conclusions

Using Hitmap abstractions greatly simplifies the writing and maintaining of a parallel program, comparing with manually hard-wiring the partition and communication structures into the code.

In PETSc the partition and communication structures are hidden into the solver and internal data structure codes. For arrays, this encapsulation leads to very simple and efficient codes. Nevertheless, for other data structures, such as graphs, the programmer needs to manually implement most of the management of the specific data structure on top of the arrays. Moreover, adding a new solver implies to deal with the library internals to generate a complete new module. Hitmap has no internal support for solvers, the programmer has to code them using the tiling interface. For new solvers, it implies a lower development effort because the Hitmap API is designed to be used by final programmers.

Some extra code complexity in PETSc comes from the limited support of data types. The array structures in PETSc are always composed of either floating point, integer, or complex elements, depending on how the library was compiled. For example, in the FEM benchmark, to store the element positions, it is necessary to distribute an array with $3 \times n$ floating point numbers, introducing special code details to align the partition. Hitmap does not limit the datatypes the programmer can use, tiles store elements of any C type. The Hitmap implementation of the FEM benchmark uses directly a single array of C structures for the position of each point. using the same partition code as for any other data type.

In Hitmap, the partition policy and communication structures are independent from the solver codes and from the internal data representation in the tile subclass. Therefore, it can be easily changed with the plug-in system. The flexibility and versatility of the Hitmap approach allows the programmer to introduce new data-structures, partitions, or program communication structures, with minimum impact on the rest of the code.

4.5.5 CSR and Bitmap comparison

To compare the efficiency of the different Shape implementations, we have executed the FEM benchmark using two hundred, randomly-generated graphs of 1 000 vertices with different density degrees as input sets. The first plot in Fig. 4.17 shows the total execution time of the FEM benchmark with respect to different density degrees. The CSR implementation outperforms the Bitmap implementation for any density degree. CSR is more efficient because it access the whole structure using iterators.

Figure 4.17 (right) shows the memory space used by the different shapes. Bitmap shapes are more efficient in terms of storage space. In addition, their access methods are also more efficient for shape-specific operations (union, intersection, adding/deleting elements), and for algorithms which performs random accesses, like some graph traversing algorithms.

Listings 4.5, 4.6 and 4.7 show excerpts of code with the structure of the CSR and Bitmap shapes, together with the code of the iterators used to traverse them. There are two macro functions for each structure type that build the loops for the row and column iterator.

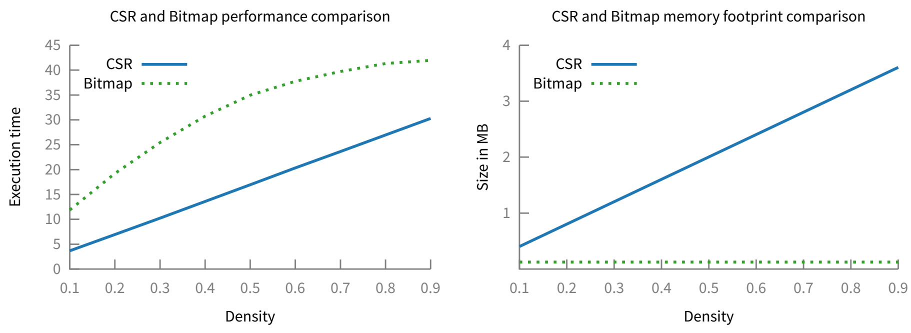


Figure 4.17: CSR and Bitmap performance (left) and memory footprint comparison (rigth) for random graphs with different densities using one node on the SC Beowulf.

The CSR iterator simply accesses the row indices array, and the compressed column array returning the current element. The Bitmap iterator has the same interface, but uses an additional inline function to locate the next active bit in the bitmap matrix.

4.6 Comparison with related work

There are many tools designed to partition sparse domains, such as Metis [80], Scotch [98], or Party [94]. These tools can be used in the context of traditional programming models like MPI [66] or OpenMP [29]. However, these models require the programmer to manually code many runtime decisions based on data partition results, adapting synchronization and/or communication to the variable sizes and inter-dependencies generated. We have integrated the Metis multilevel k-way partition technique as a new layout plug-in in Hitmap. Other techniques can also be added, offering a unified interface to perform partitions in sparse domains.

Regarding specific parallel libraries for sparse domains, such as OSKI [128] or PETSc [13], they provide frameworks with extremely efficient kernels and solvers for a great variety of linear algebra problems. The parallel strategies of these solvers have been defined specifically and they are hard-coded inside the tools of each particular framework. Therefore, a deep understanding of the framework internals is needed to change them, either to add new parallel algorithms or strategies, or to optimize them for new architectures.

PETSc and Hitmap also present several differences. First, PETSc offers only one partitioning scheme where rows are distributed among processors [13]. On the contrary, Hitmap can be extended by the user with new partition methods. For example, we currently support techniques such as multidimensional block or cyclic partitions. Second, PETSc is designed

```c
/**
 * CSR sparse shape, implements HitShape
 */
typedef struct{
    int cards[HIT_MAXDIMS];          /**< Cardinalities      */
    idxtype * xadj;                  /**< Row indices        */
    idxtype * adjncy;                /**< Compressed columns */
    HitNameList names[HIT_MAXDIMS]; /**< Name lists         */
} HitCShape;

/**
 * CSR row iterator
 */
#define hit_cShapeRowIterator(var, shape) \
    for(var=0; var<hit_cShapeCard(shape, 0); var++)

/**
 * CSR column iterator
 */
#define hit_cShapeColumnIterator(var, shape, row)    \
    for(                                              \
        var = hit_cShapeFistColumn(shape, row);      \
        var < hit_cShapeLastColumn(shape, row);      \
        var++                                         \
    )
```

Listing 4.5: CSR structure and iterators.

```c
/**
 * Bitmap sparse shape, implements HitShape
 */
typedef struct{
    int cards[HIT_MAXDIMS];          /**< Cardinalities. */
    int nz;                          /**< Non-zero elements. */
    HIT_BITMAP_TYPE * data;          /**< Bitmap array */
    HitNameList names[HIT_MAXDIMS]; /**< Name lists */
} HitBShape;

/**
 * Bitmap row iterator
 */
#define hit_bShapeRowIterator(var,shape) \
    for(var=0; var<hit_bShapeCard(shape, 0); var++)

/**
 * Bitmap column iterator
 */
#define hit_bShapeColumnIterator(var,shape,row)              \
    for(                                                     \
        var=hit_bShapeColumnIteratorNext(shape, -1, row);    \
        var<hit_bShapeCard(shape,1);                         \
        var=hit_bShapeColumnIteratorNext(shape, var, row)    \
    )
```

Listing 4.6: Bitmap structure and iterators (part 1).

```c
/**
 * Return the next column (next non-zero element of the bitmap)
 */
static inline int hit_bShapeColumnIteratorNext(HitShape shape, int var){

    size_t i;

    // 1. Index of the element and Offset of the bit in the element
    size_t xind = hit_bitmapShapeIndex(var);
    size_t xoff = hit_bitmapShapeOffset(var);

    // 2. Check if there is a 1 bit in the current element
    HIT_BITMAP_TYPE element = hit_bShapeData(shape)[xind];
    HIT_BITMAP_TYPE mask = HIT_BITMAP_1 >> xoff;
    for(i=0; i<HIT_BITMAP_SIZE-xoff; i++){
        if( (mask & element) != 0 ){
            return var + (int) i;
        }
        mask >>= 1;
    }

    // 3. Find the next element that have 1s.
    xind ++;
    while( hit_bShapeData(shape)[xind] == 0 ){
        xind++;
    }

    // 4. We do the same as 2. to get the bit location
    element = hit_bShapeData(shape)[xind];
    mask = HIT_BITMAP_1;
    for(i=0; i<HIT_BITMAP_SIZE; i++){
        if( (mask & element) != 0 ){
            return (int) (xind * HIT_BITMAP_SIZE + i);
        }
        mask >>= 1;
    }

    // Bit not found
    return -1;
}
```

Listing 4.7: Bitmap structure and iterators (part 2).

to solve scientific applications modeled by partial differential equations. The data structures supported are vectors and matrices of basic scalar data types, implemented in opaque structures that are managed by provided internal solver implementations. Instead, Hitmap is a general-purpose library that allows generic applications to be programed with any kind of structured or dynamic data types, allowing the user to access the elements directly. This makes it possible to implement other applications, such as lexicographic sorting of strings, local DNA sequence alignment, etc., without modifying the library. In [59] we show examples of how Hitmap also supports hierarchical dynamic decompositions of virtual topologies and data structures to easily implement algorithms such as Quicksort, or some N/body interaction algorithms. Third, Hitmap integrates in a single API the partition and distribution of sparse matrices and graphs. PETSc does not currently provide tools that completely manage the migration and node renumbering, since they are dependent on the particular data structure type needed for the application [13].

There are few proposals that use an unique representation for different kinds of domains. Chapel [27], a PGAS language, proposes the same abstraction to support distributed domains for dense and sparse data aggregates. As long as the programming approach of PGAS languages hides the communication issues to the programmer, efficient aggregated communications can not be directly expressed, and most of the times cannot be automatically derived from generic codes. Specific optional interfaces should be defined for different data structures and mapping techniques to achieve good efficiency. Chapel also still lacks an appropriate support for graph structures and graph-partitioning techniques.

The original Hitmap library share some concepts with HTAs (Hierarchically Tiled Arrays) [19], a library that supports dense hierarchical tiles that can be distributed across processes in distributed- or shared-memory architectures. HTAs lacks support for sparse domains. It presents a limited set of regular partitioning and mapping functionalities, and communications are dependent on them.

4.7 Conclusions

In this chapter we present an approach to integrate dense and sparse data management in parallel programming. With this approach it is possible to develop explicit parallel programs that automatically adapt their synchronization and communication structures to dense or sparse data domains and their specific partition/mapping techniques.

We have shown how the proper abstractions to support this approach in Hitmap can be introduced. We have augmented our runtime with a common interface that integrates dense and sparse data management. With it, our generic parallel system can be use to program different types of applications regardless of the selected data structure. To illustrate how the approach and the tool work, we have developed three different applications and we have compared their performance and programming effort with respect to a manually-developed

MPI code, and to an implementation that uses PETSc, a state-of-the-art tool for parallel array computations. Our experimental results show that the abstractions introduced in the library do not lead to parallel performance penalties, while greatly reducing the programming effort.

Chapter 5

A portable dataflow model and framework

As Chapter 2 described the most common parallel programming libraries and tools are currently based on message-passing for distributed-memory, or thread models for shared-memory environments. Using these models for programming dynamic and dataflow applications is challenging, especially in hybrid distributed- and shared-memory platforms.

Supporting these kind of applications is one of the steps considered in this work towards the development of a complete runtime system for a parallel framework. Thus, in this chapter we introduce a new programming model based on the dataflow paradigm, where different activities can be arbitrarily linked, forming generic but structured networks that represent the overall computation.

We present two iterative refined model proposals, and their corresponding prototype frameworks. The final framework transparently supports hybrid shared- and distributed-memory systems, using a unified one-tier model. The implementation layer is based on a Multiple-Producer/Multiple-Consumer channel system with distributed work-stealing, that works independently of the mapping of the activities to the available computing resources. We also discuss how this kind of models can be used as a programming layer to implement generic parallel skeletons.

5.1 Introduction

Dealing with hierarchical combinations of parallel paradigms leads to highly complex codes, with many decisions about mapping hard-wired into the code by the programmer. Thus, new parallel programming models with a higher level of abstraction are needed to deal with the new complex, bigger, and changing systems.

Stream and dataflow libraries and languages, such as FastFlow [5], OpenStream [103], or S-Net [64], are a promising approach, because they offer high level models that allow the computational elements and its dependencies to be defined separately. In this way, the programmers can focus on the definition of the problems, leaving the responsibility of executing the program in parallel to the runtimes. However, these models either lack a generic system to represent MPMC (*Multiple-Producer/Multiple-Consumer*) channels with optional loops, free from concurrent problems, or lack mechanisms to intuitively express task-to-task affinities, which would allow a better exploitation of data-locality across state-driven activities.

In this chapter, we propose a new parallel programming model based on dataflow computations. This model will be a supplement to the static communication structures available in Hitmap, to produce a complete runtime system for generic types of parallel applications. We present two iterative proposals. The first version of the model is based on Petri nets [1, 95], a well-known and established formalism for modeling and analyzing systems. This first model represents the task flow with two simple element types (processes and containers). The model supports both static and dynamic structures. We discuss how this model can be used as a programming layer to implement generic parallel skeletons.

In a second approach, we refine the previous work with an application of Colored Petri nets [77]. This new model introduces a generic form of describing a program as a reconfigurable network of activities and typed data containers arbitrarily interconnected. It presents an abstraction for an MPMC channel system that also includes a work-stealing, load-balancing mechanism. We introduce structured programming restrictions which ensure that computations terminate and are dead-lock free. It can be checked, at initialization time, whether the network complies with these conditions. For non-structured networks, classical Petri net analysis techniques can be used to try to detect deadlocks and traps. The prototype of the proposed model also allows the programmer to set task-to-task affinity in order to exploit data locality. The techniques or policies used to map the activities to the available resources are completely independent from the system description. It uses a single representation mechanism for shared and distributed memory models.

We present an evaluation of our proposal using examples of four different application classes. We describe how they are represented in our model, showing how to express different types of parallel paradigms, and static and dynamic synchronization structures. Moreover, experimental work has been carried out to show that the programs generated using our framework achieve good performance, comparing with FastFlow [5], another state-of-the-art tool, or in comparison with manual implementations using message passing. These experiments show that the overheads introduced by the new abstractions do not have a significant

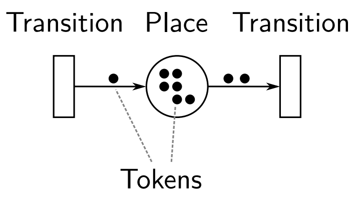


Figure 5.1: Example of a Petri net with two transitions, one of them sending tokens to a place and the other one retrieving them.

impact. Finally, an analysis of different programming effort metrics shows that the use of the framework reduces the programming cost comparing with other approaches.

5.1.1 Brief introduction to Petri Nets

Petri nets [95] are a mathematical modeling language for the description of systems. It is a particular kind of directed bipartite graph, whose nodes represent transitions. A Petri net graph has two kinds of nodes, called *places* and *transitions*. The contents of the places represent the state of the system, while the transitions are the events that can take place. They are connected using *arcs* that go from a place to a transition or from a transition to a place. Each place can be marked with one or more tokens. When a transition is fired, it removes tokens from its input places and adds tokens to its output places. Using a mathematical notation, a Petri net graph is a tuple:

$$PN = (P, T, F, M_0)$$

where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, and $M_0 : P \to \mathbb{N}$ is the initial marking expressing the distribution of tokens over the places.

Petri nets are usually represented using a graphical notation where places are drawn as circles and transitions are drawn as rectangular boxes. The directed arcs connect places and transitions. Finally, the marking is drawn as dots inside the places. Figure 5.1 shows an example of a Petri net with two transitions connected by a place.

Colored Petri nets [76, 77] is an extension of the original Petri nets for constructing models of concurrent systems. In short, Colored Petri nets extend the original model using primitives for the definition of data types to distinguish between tokens, and add a functional programming language to describe data manipulation inside the transitions.

5.2 A first approach to a dataflow model

In this section we present the first approach of our dataflow programming model. The model is based on regular Petri nets. We will add new concepts to the original Petri nets definition to describe the tasks.

The top level element of the model is an *application*. It corresponds to a Petri net, and it is defined as a 3-tuple, $A = (C, P, F)$ where:

- A *task* is a unit of data of a particular type that is processed by the application. It extends the concept of token in a Petri net, adding type and value.

- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of *task containers*. This is one of the partitions of the bipartite graph. The task containers correspond to the Petri net *places*, although task containers are typed and store tasks instead of tokens. The task type has to agree with the container type.

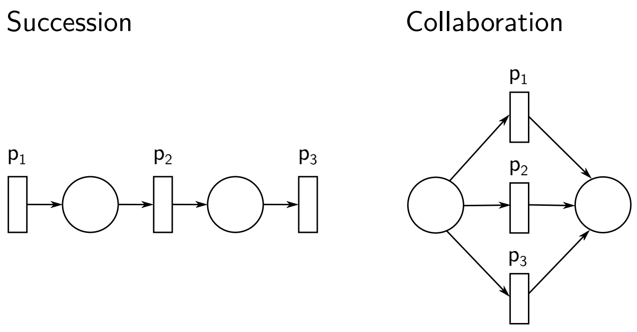
- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *processes*. They are the equivalent to the Petri net *transitions*. This set of process is the other partition of the bipartite graph. A process executes a function, they are defined by the programmer using a sequential programming language. The function has r inputs and s outputs: $f_i : x_1, x_2, \ldots, x_r \rightarrow y_1, y_2, \ldots, y_s$

- The last element of the application $F \subseteq (P \times T) \cup (T \times P)$, is a set of *flow relationships* (Petri net arcs) between task containers and processes, and vice versa, defining the edges of the bipartite graph.

A container is called an *input container* for a process if there is an arc from it to the process. Analogously, a container is called an *output containers* if there is an arc from the process to that container.

An application net can be nested inside a process node. Transient and persistent nesting modes (see Sect. 2.4.3) can be represented with this model. In transient nesting, a process will execute another application as part of the function, while in persistent mode a process can be replaced by another net, keeping its inputs and outputs.

We define two operators that help to define the structure of the application: *Succession* (•) and *collaboration* (||). The succession operator links several processes one after the other using containers and creates the flow relationships between them. The collaboration operator creates a different structure where the processes share the input and output containers. They collaborate to solve the whole problem by competing for tokens in the input containers. Figure 5.2 shows the representation of the composition operators.



$$p' = p_1 \bullet p_2 \bullet p_3 \qquad p' = p_1||p_2||p_3$$

Figure 5.2: Representation of the composition operators.

5.2.1 Execution semantics

Once created, the structure of an application is fixed, although its state (the distribution of tasks in the containers) can change. The behavior of the application is described in terms of those states. The tasks in the containers are consumed or generated by the processes based on the following rules:

- At the initial state of the application, all the containers are empty. There is not initial marking.
- When a process is executed, it consumes tasks from each of its input containers. There should be at least one task at each one of them to execute the process.
- The retrieved tasks are fed to the process function. The result tasks are sent to the output containers. The process function may produce tasks for none, or for any number of its output containers.
- If a process has no input containers, its function will be executed once.
- The evolution of the application is not deterministic. When more than one process could be executed, it is not determined which one will be executed first.
- The execution finishes when all the tasks in places that can imply a process execution have been processed.

5.2.2 Representing skeletons with this model

We discuss in this section how algorithmic skeletons can be represented using processes and containers. We have selected one representative example from each group of skeletons defined in the taxonomy of Sect. 2.4. Figure 5.3 shows the structure for each one of the examples. The figure uses the standard Petri net representation, where a circle indicates a container, a process is shown as a rectangle, and the flow relationships are arrows from/to the elements.

Pipeline A Pipe skeleton is composed of a set of connected stages. The output of one stage is the input of the following one. The structure of this skeletons is just a set of processes (one for each stage) that exchanges tasks using containers. As shown in Fig. 5.3, a process can receive tasks from several stages using different containers, leading to a more complex pipeline structure. In the same way, a process can feed tasks to more than one output container.

Farm A Farm skeleton, also known as master-slave, or master-worker, consists of a farmer and several workers. The farmer receives a sequence of independent tasks and schedules them across the workers. In a farm skeleton structure, the farmer and the workers are independent processes. There exist two tasks containers. The first one is shared by all the workers and it keeps the tasks that are ready to be scheduled to them. The other one is used to store the output results. In some farm configurations, the workers can add more tasks to the input container. This happens in the example shown in Fig. 5.3, where there are backward arrows from workers to the input place.

Stencil Although the model is more adecuated to represent dynamic or dataflow guided skeletons, it can also represent static communication structures with predetermined synchronization behavior. A Stencil skeleton updates the value of each element of a data structure applying an operation with the values of their neighbor elements. Figure 5.3 shows an example of an 1D stencil. The structure that represents this skeleton has a container for its local elements and two containers for the values of the neighbors. Each process updates its local part and inserts the values needed by its neighbors in the appropriate containers.

Map-Reduce This is a distributed programming pattern used for efficient large-scale computations that proposes two steps: map and reduce. The computation in the map step takes a set of input key/value pairs and processes them in parallel. The result for each pair is another set of intermediate output key/value pairs. The reduce step merges together all the intermediate pair associated with the same key, returning a smaller set of output key/value pairs. A Map-Reduce structure has a pair of process sets, one with the processes performing the map operation and the other performing the reduction. They are connected by several task containers that hold the intermediate key/value pairs.

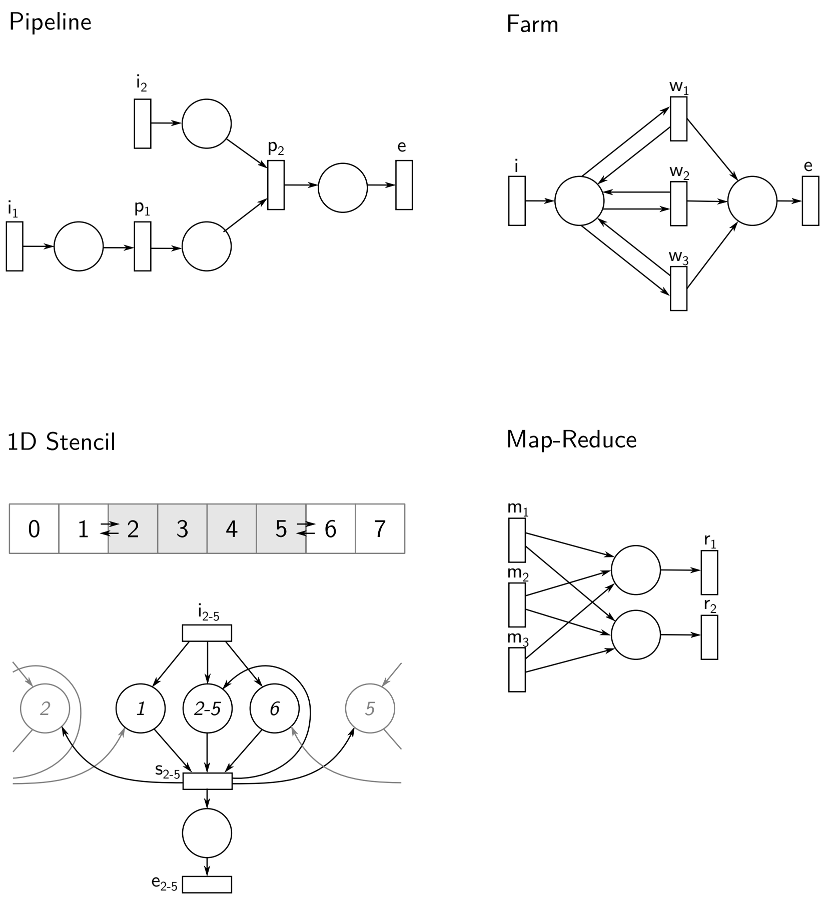


Figure 5.3: Common skeletons using the model. A circle represents a container and a rectangle indicates a process.

5.2.3 First model prototype

The prototype of the first version of the model has two main classes: A `HitTask` and a `HitWorker`. A HitTask class, an abstract datatype, is used to encapsulate the data that flows between application stages. A HitWorker is a generic worker that manages the data communications and controls the execution of the user function.

The containers of the model are implemented as lists of tasks inside the workers. When a worker generates a task for another process, the task is sent using MPI communications, and it is inserted in the worker task list. The arcs of the bipartite graph are task channels between workers, creating successor-predecessor relations.

5.2.4 Implementation example using the first model

This section exemplifies how to create skeletons with our implementation of the model. Listing 5.1 shows an example of a pipeline. Each stage of the pipeline is implemented as a function with the prototype shown in Line 2. A new task type is declared at Line 5. Line 8 contains the function for the middle stage that will be executed by a `HitWorker`

The function in Line 24 creates a three stage pipeline. It receives three function pointers as parameters, one for each stage. These user functions must receive a `HitWorker` structure pointer. The `pipe3` function creates three workers with the corresponding user function and defines the number and types of the inputs and outputs. Then, it combines the workers with the succession operator to create the relationships. The result is another worker that encapsulates the previous ones We can use it to execute the whole pipe with a single call in the main function (Line 38).

Creating a farm is a similar process but uses the collaborator operator instead. More complex computation structures can be created manually defining how the inputs and outputs of the different workers are linked.

5.2.5 Limitations

This first approach to create a dataflow programming model has some limitation. Its original goal was to allow to implement common parallel skeletons, so it supports many types of usual parallel patters, but it is not generic enough to implement every type of application. For example, the model cannot be used to represent general applications with loops. Moreover, the current prototype does not support hierarchical composition. The next section describes our second version of the model, that extends this first approach using Colored Petri nets [77]. It can be used to build more generic networks.

```c
// Typedef for the user function
typedef void (*HitWorkerFunc) (HitWorker*);

// Create the double task type
hit_taskNewType(double);

// Function for the middle stage
void process(HitWorker * worker){

   HitTask_double intask;
   HitTask_double outtask;
   hit_taskAlloc(&outtask,double);

   while(hit_workerGetTask(worker,&intask)){
      hit_taskData(&outtask) = hit_taskData(&intask) * 2.0;
      hit_workerSendTask(worker, &outtask);

      hit_taskFree(&intask);
   }
   hit_taskFree(&outtask);
}

// Three-stage pipeline
HitWorker pipe3(HitWorkerFunc f_ini, HitWorkerFunc f_mid, HitWorkerFunc f_end){
   HitWorker pipe;
   HitWorker * workers = malloc(3 * sizeof(HitWorker));

   hit_workerCreate(&workers[0], f_ini, 0, 1, HIT_DOUBLE);
   hit_workerCreate(&workers[1], f_mid, 1, 1, HIT_DOUBLE, HIT_DOUBLE);
   hit_workerCreate(&workers[2], f_end, 1, 0, HIT_DOUBLE);

   hit_workerOpSuccession(&pipe,3,&workers[0],&workers[1],&workers[2]);

   return pipe;
}

// Main function
int main(int argc, char ** argv) {

   hit_comInit(&argc,&argv);
   HitWorker pipe = pipe3(init,process,end);
   hit_workerExecute(&pipe);
   hit_comFinalize();
}
```

Listing 5.1: Fragment of code showing the creation of a pipe using the first prototype.

5.3 The Hitmap++ model

This section describes the details of our final solution to create a generic dataflow programming model. This model is based on Colored Petri nets [76, 77]. We have also created a framework called Hitmap++ that implements the proposed model. Hitmap++ is a parallel programming framework implemented in C++ that exploits dataflow parallelism for both shared- and distributed-memory systems making the program correctness independent of the mapping policy applied at runtime.

The Hitmap++ model allows generic networks of computational elements (transitions) to be created that produce or consume tasks (tokens) which are stored in shared containers (places). A transition takes one token from each one of its inputs and performs some activity with them. It may then add tokens to any/all of its output places. This activity is repeated while there are tokens in the input places.

5.3.1 Mode-driven model formulation

We propose the computation inside the transitions to be mode-driven. Tokens are colored according to the mode in which the transition that produces them is executed. Transitions read tokens from their input places with the color of the mode they are executing. Special colored tokens can be used to signal a mode change to other transitions (*mode-change signal*). The different activities associated which each mode inside the same transition are mutually exclusive. A transition changes its mode when the mode-change signal have been received in all its input places, and from all the transitions connected to those places, indicating that no producer can still send tokens with the previous mode color. Tokens generated in a previous mode are always consumed before the mode-change signals of the corresponding mode. Then, the signals are processed to select the new mode activity. The change of mode in a transition automatically produces mode-change signals on all its output places. Thus, signals are propagated automatically across the network, flushing tokens produced on the previous mode, before changing each next transition to the new mode. A special mode-change signal is used to indicate the execution termination of the transitions across the network. Modes can also be used to dynamically reconfigure the network to perform different activities. A transition mode enables a set of arcs. A mode change, which happens after all the tokens from the previous mode have been consumed, replaces the set of active arcs. An example of a network with modes can be seen in Fig. 5.4. The network has a transition (A) that sends tokens to another transition (B) with two modes. On each mode, the transition B will send tokens to a different destination, C or D. When B receives the first signal, it changes from sending tokens to C to send tokens to D.

Finally, the modes can be used to define task-to-task affinities between the mutually-exclusive transitions, to enable data locality. Two mutually-exclusive activities may be associated in the same transition if the aim is to execute them in the same thread. This allows them to share data structures.

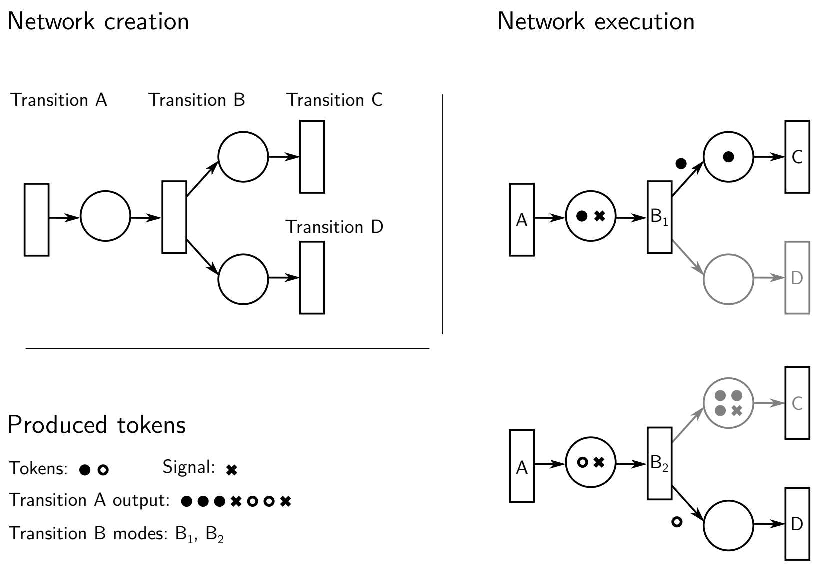


Figure 5.4: Example network with a transition that sends tokens to another two transitions depending on the mode.

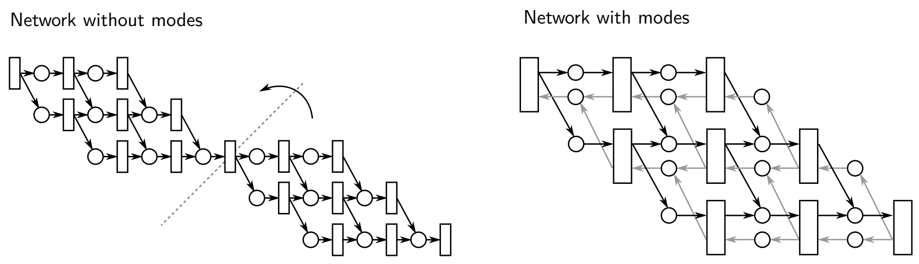


Figure 5.5: SmithWaterman network structure with and without modes.

For example, data affinity is used in the Smith-Waterman algorithm, which is one of the benchmarks discussed in the experimental section. This benchmark performs a two-phase wavefront algorithm. In the first phase, it calculates the elements of a matrix starting from the top left element. The second phase is a backtracking search that starts from the bottom right element and uses the data obtained from the previous phase. As is shown in Fig. 5.5, it is possible to create a network to model this kind of problem without using the modes. However, using the modes, we can fold that network, adding two different activities in the transitions. Thus, each transition can perform the two required activities with its own portion of the matrix, avoiding communications of matrix portions that imply sending big tokens through places.

5.4 Programming with Hitmap++

We have developed a prototype of the framework to implement parallel programs in accordance with the proposed model. The current prototype relies on POSIX Threads Programming (Pthreads) and the standard Message Passing Interface (MPI) to support both shared- and distributed-memory architectures. This section explains the key features of the programming framework. It contains a summary of the Hitmap++ API, a description how to build a program network, and details about the mode semantics. The main Hitmap++ classes are shown in the UML diagram in Fig. 5.6, while Tables 5.1 and 5.2 describe the main API methods.

5.4.1 Building transitions

To use this framework, the user has to create a class that extends the provided `Transition` class with the sequential parts of the program (see examples in Listing 5.2 and Listing 5.3). The `init` and `end` methods can be extended to execute starting and ending actions before and after the execution of the program. The user classes should introduce one or more new methods with arbitrary names to encapsulate the code for a particular mode activity. The association between modes and activity methods is established when building the network (see Sect. 5.4.2).

The activity method is automatically called when there is at least one token on each of the input places. If no input places have been defined for a particular mode, it will be called just once. The user-defined activity methods can use the `Transition::get` or `Transition::put` methods to retrieve tokens from, or append tokens to, the current input or output places active in the current mode. The `get` method retrieves one token for each of the active input places of the current mode. The `put` method adds a token to a specific output place. The output place can be selected by its identifier using the second argument of the `put` method. It can be omitted if there is only one active output place in the mode. On each activity method invocation, Hitmap++ ensures that the `get` method can be called once. Additional calls to `get` will wait until there is at least one token in each input place once more. If the framework detects that the producers have ended their modes, meaning no more tokens will be produced, the call to `get` will throw an exception.

A mode finishes when the producer transitions have sent a mode-end signal indicating that they have finished the activity in that mode, and all the tokens in the places, that were generated in the previous mode, have been processed. At this moment, the transition automatically evolves to the next programmed mode and the flow process starts again. The `Transition::mode` method can be called at any time to select which will be the next mode activated in this transition when a mode-change situation arises.

Apart from the user-defined modes, there are two special mode identifiers, *END* and *RESTART*. The *END* identifier, which is the default next mode, unless changed by the programmer, indicates that when this mode finishes, the computation on this transition ends. Using the *RESTART* identifier is the same as calling the `mode` method with the current mode as argument.

The example in Listing 5.2 extends the `Transition` class by declaring a user activity method. The method retrieves a token from one place, processes it, and sends the result to an output place. In Listing 5.3, there is a more complex example. The method retrieves one token from each of the two input places two times, each one of a different type. Depending on the result computed with their values, the method sends a token to one of the two possible output places.

The tokens are C++ variables. They could be of any type since they are handled using template methods. The marshaling and unmarshaling is done internally with MPI functions. The basic types (char, int, float, ...) are enabled by default. User defined types require the programmer to declare a data type with a Hitmap++ function (`hitTypeCreate`) that internally generates and registers the proper MPI derived type using a similar approach than Hitmap to create communication structures for tile types.

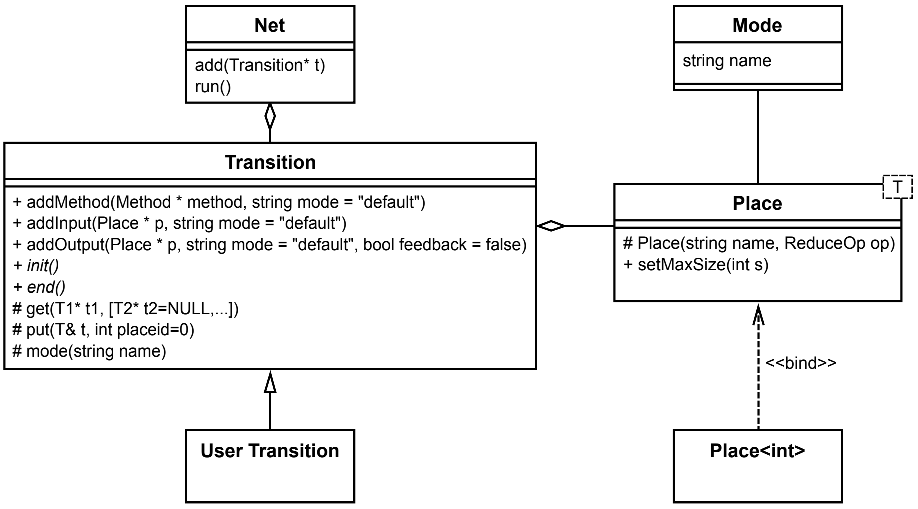

Net
add(Transition* t)
run()
Mode
string name
Transition
+ addMethod(Method * method, string mode = "default")
+ addInput(Place * p, string mode = "default")
+ addOutput(Place * p, string mode = "default", bool feedback = false)
+ init()
+ end()
# get(T1* t1, [T2* t2=NULL,...])
# put(T& t, int placeid=0)
# mode(string name)
Place
T
# Place(string name, ReduceOp op)
+ setMaxSize(int s)
<<bind>>
User Transition
Place<int>

Figure 5.6: UML diagram of the framework.

```cpp
class MyTransition: public Transition {
public:
    void execute(){     // User activity method
        double intask;
        get(&intask);   // Retrive a token from the place
        double outtask = process(intask)
        put(&outtask);  // Put the token into the output
        mode(END);      // Select the next mode
    }
};
```

Listing 5.2: Hitmap++ code example of the creation of a transition extending the basic Transition class.

```cpp
class MyTransition2: public Transition {
public:
    void execute(){
        double d1, d2; int i1, i2;
        get(&d1, &i1);   // Get one pair of tokens
        get(&d2, &i2);   // Get other pair

        double result =  process(d1,d2,i1,i2);
        // Send a token to a particular place
        if(result > 0)
            put(&result,"place1");
        else
            put(&result,"place2");
    }
};
```

Listing 5.3: Example using the active method and the get method with several inputs.

Object	Method	Description
Transition	`addMethod(method* m, string m)`	Sets the user activity method for a mode. The activity method will be called when the transition has tokens to be processed. If no mode is selected, the provided user method will be set as the handler of the default mode.
	`addInput(Place* p, string m)`	Adds a place as input for the selected mode. This method can be called multiple times to add additional input places.
	`addOuput(Place* p, string m, bool feedback)`	Adds a place as output for the selected mode. The optional parameter feedback is used to indicate a feedback-edge. This method can be called multiple times to add additional output places.
	`init()`	Virtual method that is called at the start of the network execution. The user can implement this method in the derived transition class to initialize internal objects.
	`end()`	Virtual method that is called at the end of the network execution. The user can implement this method in the derived transition class to free internal objects.
	`get(T1* t1, [T2* t2, ...])`	Template method to retrieve tokens from the input places. It receives a pointer to the received token for each input place in the current mode. It is overloaded to support any number of input places.
	`put(T& t1, int placeid = 0)`	Template method to store a token in an output place. This methods accepts a pointer or reference to the token and the index or name of the chosen output place. The default is the place at index 0.
	`mode(string name)`	Method to select the next active mode. The mode change will occur when all the tokens in the current mode have been processed.

Table 5.1: Description of the Hitmap++ API (part 1).

Object	Method	Description
Place	`Place(string name = "", ReduceOp op = Null)`	Constructor of the Place class, it has two optional arguments: its name and a reduce operator to create a reduce place (see Sect. 5.4.5).
	`setMaxSize(int s)`	Sets the size of the place. This value defines the granularity of the internal communications, it is the number of packed tokens that will be transferred together.
Net	`add(Transition* t)`	Adds a transition to the network.
	`run()`	Initializes the internal communication structures and starts the execution of the transitions of the network.
HitType	`HitType* hitTypeCreate(class, members, [name, size, type], ...)`	Creates a new HitType object. It receives the name of the class, its number of members, and a triplet (name, size, and type) of each member. It constructs the internal communication types.
	`setMaxElems(string array, size_t size)`	Sets the maximum number of elements for a hitarray member. This method receives the name of the hitarray member in the HitType.

Table 5.2: Description of the Hitmap++ API (part 2).

```
 1  Place<double> placeA, placeB;  // Declare the places
 2  placeA.setMaxSize(10);      // Set the place size
 3
 4  MyTransition transition;
 5
 6  // Add the method and places to the default mode
 7  transition.addMethod(&MyTransition::execute);
 8  transition.addInput(&placeA);
 9  transition.addOutput(&placeB);
10  ...
11
12  Net net;                // Declare the net
13  net.add(&transition); // Add the transition
14  net.run();              // Run the net
```

Listing 5.4: Hitmap++ example of the network creation.

5.4.2 Building the network

Once the transition classes are defined, the programmer should proceed to build the network in the `main` function. This implies creating transition and place objects, associating the activity methods, input, and output places to modes on the transitions, and finally adding the transitions to a Net object.

Listing 5.4 shows a simple code with a network using the previously shown `MyTrans-ition` transition from Listing 5.2 The code shows how the user method and two places are added to the default mode of the transition, how a transition is added to a network, and the method call that starts the computation.

The first step is to create the places that will be used in the application (Line 1). The `Place` class is a template class used to build the internal communication channels. The size of the place defines the granularity of the internal communications: It is the number of packed tokens that will be transferred together. The user must set it in accordance with the token generation ratio of the transition.

The next step is to set the activity method and the inputs and outputs for each mode. The `addInput`, `addOutput`, and `addMethod` methods have an optional parameter to specify the mode. When this parameter is not specified, a default mode is implicitly selected. Lines starting at 7 set the activity method, an input place, and an output place for the default mode. Multiple calls to the `addInput` or `addOutput` for the same transition mode, allow MPMC constructions to be built.

Finally, all the transitions are added to a `Net` class that controls the mapping and the execution (Lines 12 and 13). Line 14 invokes the `Net::run` method that starts the computation.

5.4.3 Structured network programming

Hitmap++ allows generic networks to be built with all the expressive power of Petri nets. In this section, we introduce structured data-flow programming conditions that restrict the network structures that can be built, in order to ensure termination and dead-lock free computations. It is possible to check, at initialization time, if the network built by the programmer complies with the structural conditions discussed below.

Data-flow networks are built as graphs. They can form fully connected graphs or not. In the last case, there are several parallel computations that are mapped simultaneously to the available resources, but they are independent and evolve at their own pace. They can be analyzed separately.

Single mode DAG networks

Networks can be built as DAGs (*Direct Acyclic Graphs*), where tokens and signals flow from sources to sinks. In this case, and for a single mode, the Hitmap++ implementation ensures that tokens will eventually flow through places. The mode-change signals, which are generated to indicate termination at the sources, flow through the network, forcing the other transitions to flush their input places, processing the remaining tokens before ending their computation. Even in the case where one or more sources do not produce any token, the end of their execution method sends the mode-change signals that flow through the network, ending the rest of the computation.

Cycles and feedback-edges

We call *feedback-edges* to those connections from a transition T to an output place S, such that there exists a path from S to T, forming a cycle in the graph. Feedback-edges present a special situation for termination (mode-change) detection. Notice that termination in Petri nets is detected by a global condition; there are no tokens in any place and all activities have finished. In Fig. 5.7, we show an example of a simple network with a feedback-edge. When termination (mode-change) signals arrive at S, the transition T should establish if there are still tokens inside the cycle that can generate new tokens back to place S. When S detects the termination (mode-change) signals in its input places, a new colored signal token should be automatically sent into the cycle by S (*feedback signal*). When this signal token is propagated through the cycle and arrives back at the place S through the feedback-edge, if no new normal tokens have been introduced in the cycle meanwhile and the input places of S contains only the mode change signals, T can terminate and propagate its normal mode-change signal. All this operations should be done automatically by the implementation transparent to the programmer.

To ensure that the new colored token that verifies the emptiness of the cycle is not propagated unnecessarily to other parts of the network, and that no tokens coming from outside of the cycle can enter after the feedback signal, the cycle subgraph has to verify the following structural conditions: (1) it is a fully connected subgraph; (2) it has only one source transition T; (3) all paths that start at T arrive at a transition F with a feedback-edge to the place S; and (4) there is no path from a transition T' outside the cycle subgraph, to any F transition that does not include transition T. The transitions F that close the cycle should only propagate automatically the signal through the feedback-edge. Multiple feedback-edges and cycles can be hierarchically nested if the signals used to verify the cycle's emptiness have different colors (the implementation can use a feedback-edge identifier as color).

To avoid global feedback-edge detection at initialization time, the Hitmap++ interface has an optional parameter in the `Transision::addOutput` method that allows the programmer to indicate that the connection is a feedback-edge.

Multiple modes

Hitmap++ ensures termination and dead-lock free computations for networks programmed as DAGs with structured feedback-edges and a single mode. Modes can be used to overlap single mode graphs, mapping mutual exclusively transitions to the same one in different modes (recall the example in Fig. 5.5). If the transitions change their modes following a depth-first search order, the network operates virtually as the expanded network with a single mode, introducing no extra concurrency problems.

Non-structured networks

Current Hitmap++ implementation cannot ensure termination and dead-lock free conditions for other more complex constructions. For those networks, classical analysis techniques of Petri nets should be used to detect such problems and warn the programmer (see e.g. [131]). Since deadlocks and traps do not depend on the initial markings, and can be detected by a structural analysis of the network, this can be done at initialization time.

Transition correctness

To ensure that the flow of tokens works properly, transitions should comply with the following conditions:

1. Transitions should provide each mode with an activity method that executes the `get` method to read tokens at least once when it is run, if there are input places for the associated mode. The method can optionally send tokens to one or more output places of the mode without any restriction.

2. Processing ratios: Although it is not a problem at the abstraction level of the model, when transitions send output tokens to places at a higher ratio than they are read and processed by the receiving transitions, tokens accumulated in places may grow indefinitely, leading to a system crash due to memory exhaustion. This should be prevented by the programmer.

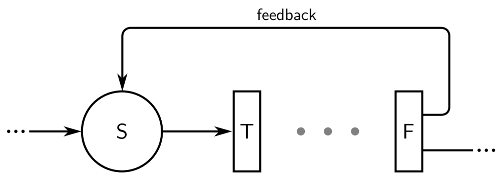


Figure 5.7: Network with a feedback-edge.

3. Paired input places: When a transition has more than one input place active in a given mode, the total amount of tokens received on each place at the end of the activity computation (before a mode-change) should be the same. In any other case, tokens received are paired until one of the input places is exhausted, with the mode-change signal active. The rest of the tokens in the other input place/s cannot be paired and processed. This is a semantical mistake. The implementation can discard the non-paired tokens with a warning, to continue processing in the new mode, but it cannot be ensured which tokens are discarded, and non-controlled stochastic behaviors can be introduced.

5.4.4 Mapping

In Hitmap++, the programs are executed using MPI. Thus, a fixed number of MPI processes will be available at launch time. At initialization time, the user builds a network with a fixed number of transitions. Thus, a mapping policy should be applied to map transition sets to the available MPI processes. MPI processes with more than one mapped transition automatically spawn additional threads to concurrently execute all the transitions. Hitmap++ implementation solves the potential concurrency problems introduced by synchronization and communication when mapping transitions to the same process (see Sect. 5.5.3). In the current prototype, the mapping policies should provide an array associating indices of transitions to MPI process identifiers. If a specific policy is not provided, there is a default fallback policy implementing a simple round-robin algorithm. Using this approach, it is possible to build mapping modules that exploit information about the physical devices to allow advanced automatic mapping policies [55].

Figure 5.8 shows an example. There is a master transition that generates tokens. Those tokens are fed to several workers, who sort the tokens into two different transition collectors. At the bottom of the figure, we can see different mappings depending on the number of nodes and processes available with the default round-robin policy.

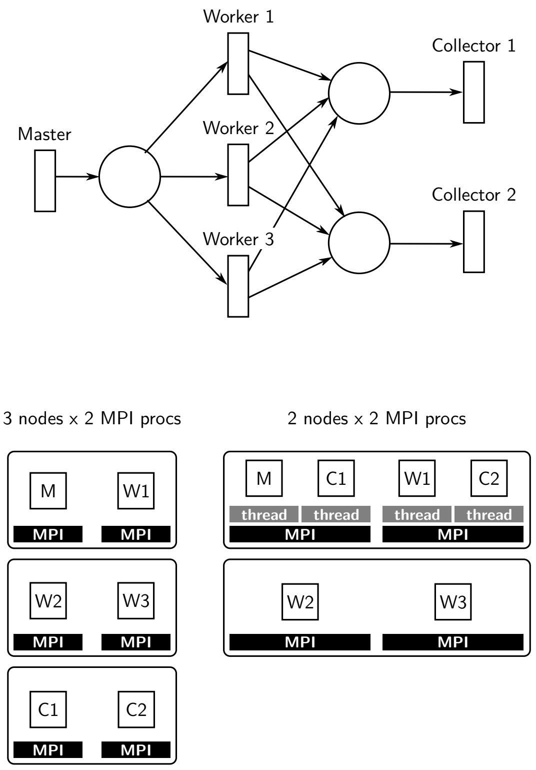


Figure 5.8: Example of a network and the default mapping in two possible architectures.

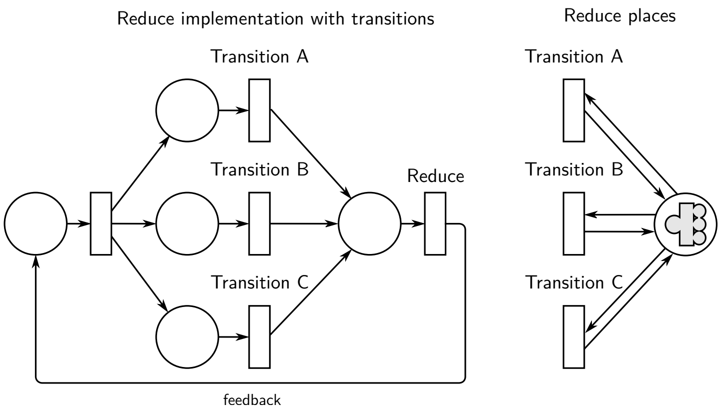


Figure 5.9: Reduce operation implemented with additional transitions, vs. implemented using a Reduce place.

5.4.5 Reduce Places

Common structured networks can be introduced as new abstractions at the programming level, bringing the possibility of using specialized and optimized implementations. For example, a common structure used as component can be an all-to-all reduce operation, where a value is obtained by combining several values with a commutative and associative operator and the result is communicated back to the transitions generating the values. This operation can be programmed with a network such as the one shown in Fig. 5.9 left.

We introduce a programming artifact named *Reduce Places* representing a small network with an implicit transition that performs a reduce operation. Instead of building a network like the one shown in the example, the programmer can use a Reduce place (Figure 5.9, right) to obtain a simpler network. Each of the involved transitions sends a token with its value and retrieves a token containing the reduced value. To create a Reduce place, the `Place` class has a constructor that receives a reduce operator as an additional argument (see API at Table 5.1).

The current implementation of Hitmap++ Reduce places performs a reduction on two levels. First, the values of the tokens from transitions running in threads of the same MPI process are reduced. Then, an MPI reduce communication operation is issued to obtain the final value. This solution allows simpler efficient networks to be implemented as encapsulated components.

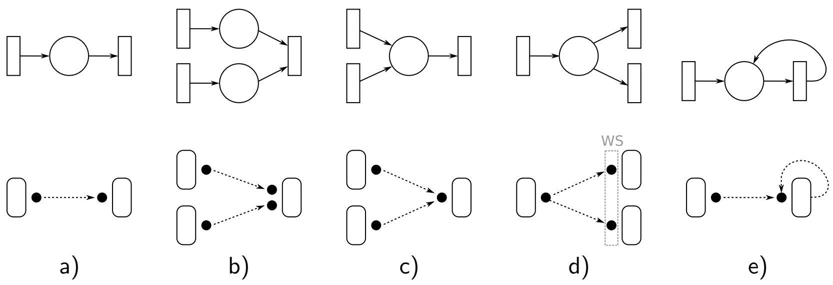


Figure 5.10: Translation from the model design to its implementation. WS: work-stealing.

5.5 Implementation details

This section discusses some of the implementation challenges associated with the model, and how they have been solved in the current prototype framework.

5.5.1 Targeting both shared and distributed systems

One of the main goals of the framework is to support both shared and distributed memory systems with a single programming level of abstraction. This is why our implementation internally uses a combination of MPI and Pthreads. The user-defined `Transition` objects that contain the logic of the problem are mapped into the available MPI processes. Since there may not be enough processes for all of the transitions, threads are spawned inside the processes if needed. Using this approach, we can easily take advantage of clusters of multicores. The data transmission is done with MPI communications, even between threads of the same processor.

5.5.2 Distributed places

The networks, composed by transitions and places, represent high level abstractions of the programs, independent of any mapping or scheduling techniques. In Hitmap++, the key that enables both a high level of abstraction and great performance is the places implementation.

The Hitmap++ places are not physically located in a single process. Instead, they are distributed token containers. The places are implemented as multiple queues of tokens located in the transitions that have been declared as using that place as input. When needed, the tokens are redistributed across those transitions, using MPI point-to-point communications. This solution builds a distributed MPMC queue mechanism that exploits data locality, and is more scalable than a centralized scheme where a single process manages all the tokens of a place. However, this is a solution that introduces some coordination challenges.

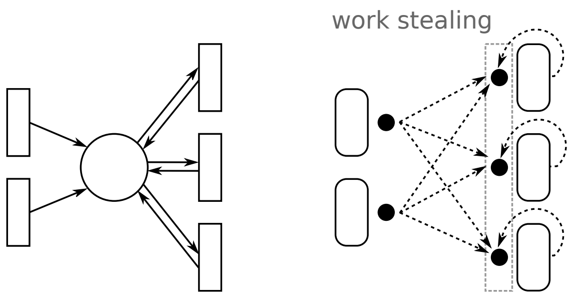


Figure 5.11: Complex network example with two producers and several workers than can also generate new tokens. The right figure represents the internal channel configuration.

The distributed places are implemented using *ports* that move the tokens following the flow of the data from the source transitions to the destination transitions. Input and output ports are linked using *channels*. Figure 5.10 shows how the arcs of the model are implemented using ports. There are five situations:

a When a place connects two transitions, a channel will be constructed to send the tokens from the source to the destination.

b When there are two or more input places in a transition, it will have several input token ports, each of them connected to the corresponding source.

c When two or more transitions send tokens to a common place, the destination will have a single port that will receive tokens, regardless of the actual source.

d If a place has several output transitions, any of them can consume the tokens. To enable this behavior, when a place is shared by several destinations, the source will send tokens in a round-robin fashion to each output port. This can lead to load unbalance if the time to consume tokens in the destinations is not compensated. To solve this, a work-stealing mechanism is used to redistribute tokens between the destination transitions.

e When a transition uses the same place as input and output, the token will flow directly to the input port for efficiency reasons.

The previous solutions can be combined to implement complex networks. The network shown in Fig. 5.11 is an implementation of a farm whose workers can generate new tasks dynamically. The work-stealing strategy ensures the load balancing.

5.5.3 Ports, buffers, and communications

The previous section showed how the translation from model places to ports and channels is done in Hitmap++. This section shows the internal port objects and explains the details about the communications and buffering.

The internal communications are handled by the objects of the class `Port`. Each transition has a port for every input or output place. When the user calls the `Transition::get` or `Transition::put` methods, those methods will use the ports to retrieve or send the tokens. The ports have a buffer where the tokens are stored. The size of the buffer is determined by the size of the place that it represents, as defined by the user with the `Place::setMaxSize` method. The size of the buffer also has an extra space for the message headers and other information that must be sent along with the tokens. When a user sends tokens to a place, they are stored in the output port buffer. The Hitmap++ runtime library decides when to perform the real communication. By default, it will try to maximize the port buffer usage, packing as many tokens as possible to minimize the number of MPI messages to be sent.

In addition to the port buffers, the transitions have queues to store the received tokens. There is a queue for each input place. Unlike the buffers, which have a limited memory space assigned, the queues grow dynamically and are only limited by the host memory. When an incoming MPI message is received, the input buffer associated to the channel is used to retrieve the tokens. After the reception in the buffer, the tokens are stored in the corresponding queue where they can be accessed by the transition `get` method. Finally, the user method is automatically called to process the available tokens when there is at least one token in each input queue.

Figure 5.12 (top) shows an example of a two-transition network. There is a producer that generates tokens that are sent to a consumer using the place A. The consumer presumably performs a filter operation on the tokens and sends some of them back to the producer using the place B. Figure 5.12 (bottom) describes the internal structures of the previous example. The producer transition (2) and the consumer transition (2') are executed in two different processes (1 and 1', respectively). Since both transitions have only one input place, they have only one input token queue (see 3 and 3'). The size of the place A is 5, thus the output port of the producer (5) and the input port of the consumer (4') have a buffer for 5 elements. In contrast, the size of B is 3, so its port buffers (4 and 5') have that same size. The figure also represents the MPI communication buffers for the two processes (6 and 6').

In the current implementation, Hitmap++ relies on buffered MPI communications. This is needed because, in some cases, the MPI implementation can decide to execute the non-buffered communications in synchronous mode. Therefore, they could cause deadlocks when there are interdependencies between paired transitions that send and receive in different modes. This could happen, for example, in the Cellular Automata benchmark that will be described in Section 5.6. If two neighbors exchange data in a synchronous mode, but there are messages in another transition mode, not yet active, both sends could be blocked. Using buffered MPI communications solves this problem.

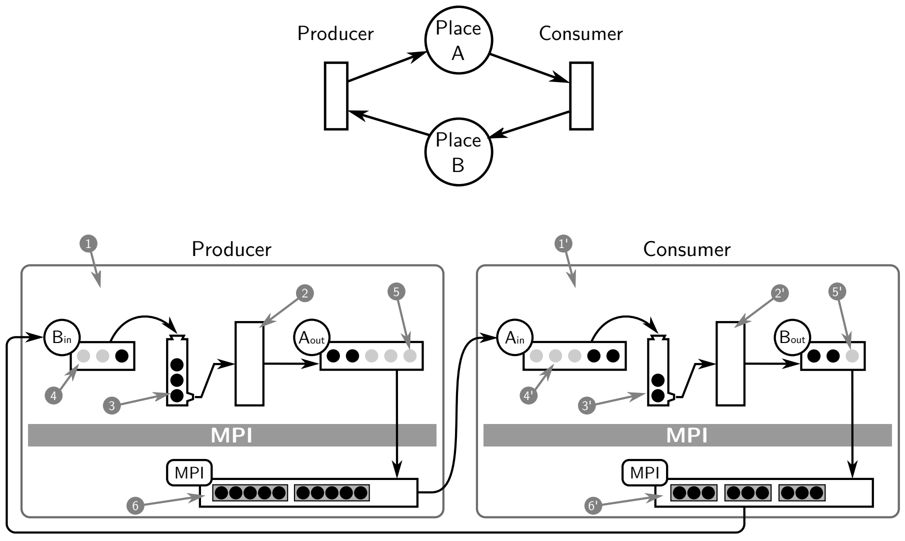


Figure 5.12: Small example network with two transitions and the description of the different buffers, data structures and control elements involved in the communications. Legend: (1) MPI process. (2) User transition object. (3) Internal token queue for the transition. (4) Input buffer port. (5) Output buffer port. (6) MPI communication buffer.

When using buffered MPI communications, Hitmap++ cannot use the default MPI buffer. Instead, MPI forces to use a new one, allocated by the framework programmer. Thus, Hitmap++ creates a communication buffer for each MPI process, taking into account the mapped transitions, their active places for each mode, and the place sizes defined by the user. With all this information, Hitmap++ calculates the upper bound of the maximum memory size to allocate the internal buffers.

With the Hitmap++ runtime, it is not possible to produce a deadlock due to port buffer exhaustion, even in unbalanced networks with cycles. Consider for example the network depicted in Fig. 5.12. Assuming that the producer and consumer send tokens with a very unbalanced ratio, causing both the port buffer and MPI buffer of the two transitions to become exhausted, it will not cause a deadlock. The runtime will keep receiving messages and storing them in the local and unlimited transition queue. Thus, the only limitation will occur when one of the processes depletes the host memory.

However, it is possible to produce a deadlock due to MPI buffer exhaustion when several transitions are mapped in the same MPI process using threads. This is a limitation of the MPI standard that only allows one communication buffer per process. Thus, it is not possible to allocate a different one for each thread, not even using individual MPI communicators. The previous example did not produce a deadlock, but if both transitions were mapped in the same process, they would share the same MPI buffer. Thus, the messages of one transition could consume all the buffer memory, preventing other transitions from performing their communications. This opens the possibility of producing a deadlock on the progression of the whole network. This can be solved by using another communication library, or a more sophisticated implementation, using for example an explicit dynamic buffer control or one-sided MPI communications.

5.5.4 Tokens: Marshalling and Unmarshalling

Hitmap++ tries to offer a high level of abstraction. Internally, it uses MPI to communicate the values of the tokens. Basic C++ types are supported by default. Thus, the programmer does not have to worry about the marshalling of classes such as `char`, `int`, or `double`. They can be used in the `Transition::get` and `Transition::put` methods out of the box.

For user-defined types, it is necessary to internally create an MPI derived datatype that will be used in the communications. Hitmap++ provides a type creation macro function (`hitTypeCreate`) that simplifies the creation of the internal MPI datatypes. Moreover, the framework eases the transmission of classes with variable length arrays as members. By default, MPI only supports struct types with fixed size and members with known offsets. We have added a new container class (`hitvector`) that mimics the interface of the C++ `std::vector` and we have added a mechanism to serialize and transfer the contents of that class using MPI. The `hitTypeCreate` macro function also creates MPI datatypes for the elements of the possible `hitvector` objects. When sending a message, Hitmap++ implementation relies on low-level pack and unpack MPI functions to do the marshalling operations.

First, it packs the content of the class members into the MPI communication buffer. Next, the number of elements of the `hitvector` objects, and finally the actual elements. The unmarshalling is done in a similar way.

An example of this functionality can be found in Listing 5.5. It contains a portion of a code implementing a mandelbrot set benchmark. Lines starting at 2 contain the declaration of the data structure used to store several lines of the grid. It has three members: the first line index, the last line index, and a hitvector with the actual elements. At Line 16, the `hitTypeCreate` macro is used to create the MPI datatype and to register it in Hitmap++. The inputs of the macro are the the name of the class, the number of members, and for each member: its name, number of elements, and type. We use the constant `hitvector`, instead of the number of elements, to indicate that a member is a hitvector. The `hitTypeCreate` macro returns a pointer to the new HitType. The method `HitType::setMaxElems` is used on the returned pointer to define the maximum number of elements of the hitvector (Line 20). Finally, Line 27 contains the code of the master transition that receives the tokens. We can send `ResultSet` objects with their variable size hitvector `elems`, and Hitmap++ library will take care of the transfer.

5.6 Case Studies: Hitmap++ Evaluation

In this section, four case studies are discussed, to test whether the model is suitable to represent different kinds of applications, to check the performance of the prototype framework, and to compare the development effort needed when using Hitmap++ with other approaches.

5.6.1 Benchmarks

Four different benchmarks have been tested. The first one calculates the Mandelbrot set, an embarrassingly parallel programming application that helps us test the basic functionalities of our proposal. Being a common example whose implementation is provided in other frameworks and tools, it also allows us to compare our implementation with other solutions.

The next two benchmarks are two implementations of a real application. They are very different implementations of the Smith-Waterman algorithm, an algorithm to perform local alignments of protein sequences. One of them is swps3 [122], a highly optimized implementation that extensively uses vector instructions. The other one, CloteSW, is a parallelization based on the implementation developed by Clote [32]. The first one is a simple task-farm application, while the second one represents a complex combination of wavefront and reduction operations.

Finally, the last benchmark is a Cellular Automata or Stencil computation. It solves the Poisson equation in a discretized 2D space using an iterative Jacobi solver. Each element in the space only interacts with its neighborhood. This kind of benchmark represents the

```
/** Class with a particular range of lines. */
class ResultSet {
public:
    int firstline;
    int lastline;
    hitvector<char> elems;

    int numlines(){ return lastline-firstline+1; }
    int size(){ return elems.size() };
    char& elem(int i){ return elems[i] };
};

/** Main function */
int main(int nargs, char * argv[]){
...
    HitType * type = hitTypeCreate(ResultSet,3,
        firstline, 1,          MPI_INT,
        lastline,  1,          MPI_INT,
        elems,     typevector, MPI_CHAR);
    type->setMaxElems("elems", cols * grain);
...
}

/** Master transition */
class Master: public Transition {
...
    void results(){
        ResultSet result;
        get(&result);
        cout<<"Received: "<<result.numlines()<<endl;
        for(int i=0; i<result.size(); i++){
            processElement(i,result.elem(i));
        }
    }
};
```

Listing 5.5: Excerpt of code from the Mandelbrot benchmark showing the token type declaration and its usage.

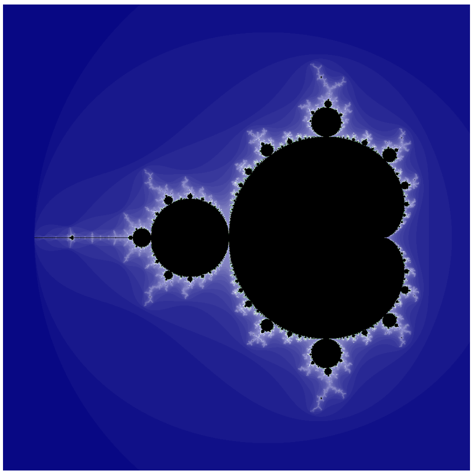


Figure 5.13: Mandelbrot set.

kernel computation in many problems usually associated with data parallelism. It has a static parallel structure, not based on dataflow parallelism.

The following paragraphs describe these benchmarks with more detail.

Mandelbrot

The Mandelbrot set [88] is one of the best-known examples of mathematical visualization. It has a very recognizable pattern (see Fig. 5.13). It has become popular as a benchmark in parallel computing since it is easily parallelizable but introduces a load-balancing problem [66].

The Mandelbrot set is defined in the following way. Given a complex number $c \in \mathbb{C}$ and the sequence $z_{n+1} = z_n + c$, starting with $z_0 = 0$, c belongs to the Mandelbrot set if, when applying the iteration repeatedly, the sequence remains bounded regardless of how big n gets.

The benchmark computes the iterative equation for each point to calculate whether the sequence tends to infinity. If this sequence does not cross a given threshold before reaching a given number of iterations, it is considered that the sequence will converge. This problem is straightforwardly parallelizable because the calculation of the equation on a particular point is independent from the result of any other point. However, it can present significant load imbalances, because some points reach the threshold after only a few iterations, others could take longer, and the points that belong to the set require the maximum number of iterations.

Smith Waterman

The Smith-Waterman (SW) solution [116] is a dynamic programming algorithm to find the optimal local alignment of two sequences. This algorithm is commonly use in bioinformatics to determine similar regions between protein sequences. It is guaranteed to find the optimal local alignment with respect to the scoring system being used. Instead of looking at the total sequence, it compares segments of all possible lengths and optimizes the similarity measure.

To find the optimal local alignment of two given sequences A and B, with lengths n and m, the Smith-Waterman algorithm uses a similarity score function $s(a_i, b_j)$ and a linear gap penalty function $w(k)$. The algorithm constructs the H matrix that contains the maximum similarity of two segments ending in a_i and b_j respectively. The first row and column of the matrix are set to 0:

$$H_{k0} = H_{0t} = 0 \quad \text{for } 0 \leq k \leq n \text{ and } 0 \leq l \leq m$$

The rest of the elements of the matrix are computed using its north, west and northwestern neighbors, as it is shown in the following equation:

$$H_{ij} = max \begin{cases} 0 & \\ H_{i-1,j-1} + s(a_i, b_j) & \text{Match} \\ \max_{k\geq 1}\{H_{i-k,j} + w(k)\} & \text{Deletion} \\ \max_{l\geq 1}\{H_{i,j-l} + w(l)\} & \text{Insertion} \end{cases}$$

Once the matrix is complete, the pair of segments with maximum similarity is found by first locating the maximum element of H. Then, a traceback procedure is used to obtain the matrix element sequence leading to that maximum value. This procedure identifies the segments as well as produces the corresponding alignment. Figure 5.14 shows an example of the Smith-Waterman algorithm.

The Smith-Waterman algorithm can be parallelized on two levels [108]. It is fairly easy to distribute the processing of each of the database sequences on a number of independent processes. On a lower level, the calculation of the H matrix elements for a given sequence can be parallelized taking into account their dependency pattern. Each element is computed using its north, west and northwestern neighbors, following a wavefront propagation.

Cellular Automata

A cellular automaton is a computational system defined as a collection of cells that change state in parallel based on a transition rule. This rule is applied to each cell and uses the cell value and the values of a finite number of the neighbors cells [114, 117]. They represent the kernel computation in many problems usually associated with data parallelism. They are a good example of Single Program, Multiple Data (SPMD) computation. They have historically been used to study a variety of problems in the biological, physical, and computing sciences.

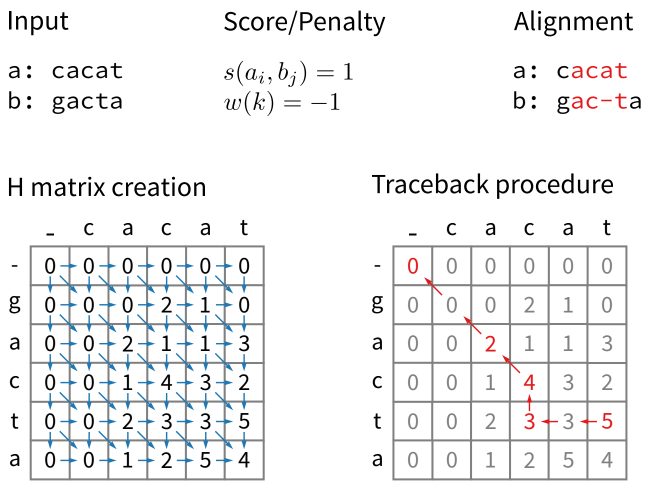


Figure 5.14: Example of the Smith-Waterman algorithm.

Cellular automata were first introduced by John von Neumann as formal models of self-reproducing biological systems. The system is composed of a grid of cells that evolve in time by applying a transition function to each cell. The function that determines the change of state is the same for all cells. The new state is based on the current value of the cell and that of its neighbors. The whole automata system does not have any input, the state of the system at time t is only determined by the state form time $t-1$.

The grid in a cellular automaton has a d-dimensional geometry. The mathematical model can define an infinite grid. However, it is easier to implement a finite grid with different boundary conditions. A grid has periodic boundary conditions if the cells at the edge of the grid have as neighbors the cells at the opposite edge. Other alternative is using fixed boundary conditions were the state of the edge cells do not change between iterations.

The neighborhood of a cell is usually its adjacent cells. However, it is possible to define different kinds of neighborhood. In a 2-dimensional grid, the two most common types are the von Neumann neighborhood (the orthogonal cells and itself) and the Moore neighborhood (the eight surrounding cells and itself).

A cellular automaton algorithm is a good candidate to be parallelized due to the high degree of parallelism present in the update function. Several considerations have to be taken into account in the implementation. First, it is necessary to analyze the data dependencies imposed by the update function. This requires different solution depending on the parallel paradigm. In a shared memory model, a double-buffering scheme is usually needed because an in-place update of the cell would destroy the value needed to update its neighbors. When

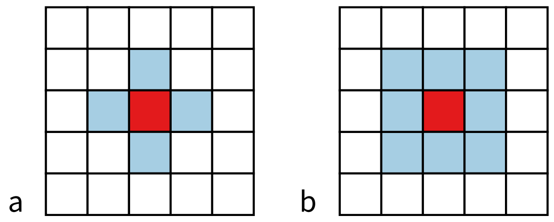


Figure 5.15: Cellular automaton in a 2-dimensional grid. (a) von Neumann neighborhood, (b) Moore neighborhood.

using distributed memory model, it is necessary to divide the grid into portions and deal with the data distribution to update each grid partition with the contribution of other parts.

5.6.2 Performance

Experimental work has been conducted to show that the implementation of Hitmap++ achieves a good performance compared with other frameworks and manual implementations. We use two different experimental platforms with different architectures: A multicore shared-memory machine and a distributed cluster of shared memory multicores:

- The shared-memory system, Atlas, has 4 AMD Opteron 6376 processors with 16 cores each at 2.3GHz, and 256Gb of RAM memory.

- The distributed system, CETA-Ciemat, is composed of several MPI nodes with two 2 Quad Core Intel Xeon processors (4 cores, 2.26GHz). Thus, each node contains 8 cores. For experimentation in this chapter, we have used 8 nodes to match the 64 cores of Atlas.

Mandelbrot set

For the Mandelbrot benchmark, we compare the Hitmap++ version against a manual MPI version, and two versions using FastFlow [5], a structured parallel programming framework originally targeting shared memory multi-core architectures with specific extensions to support combined distributed- and shared-memory platforms. We tested one shared- and one distributed-memory FastFlow benchmarks. The shared-memory one is the implementation included in the FastFlow distribution examples that uses the FastFlow farm pattern. We have developed the distributed version using the two-tier model of the extended FastFlow library that supports both shared and distributed memory using different classes [4]. This last version has a farm pattern inside each distributed node. There is also an upper tier producer that coordinates the work and feeds tasks to the nodes, with their lower tier farms.

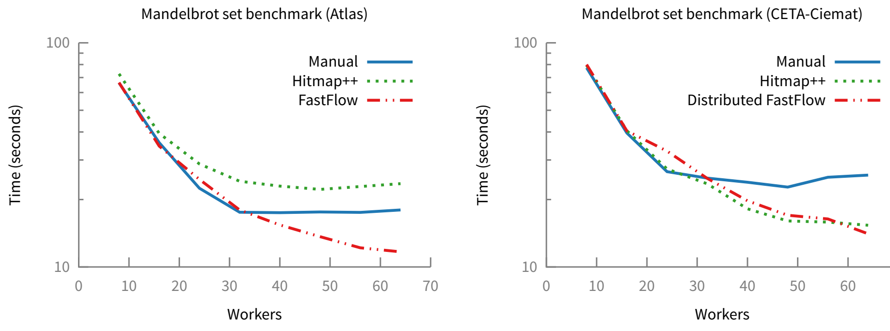


Figure 5.16: Mandelbrot set benchmark results.

All the implementations use a farm structure that processes the grid by rows. The manually developed one implements a simple farm algorithm in MPI. The Hitmap++ version uses a network with a producer transition and several worker transitions connected by a single place, This is a very simple benchmark used to test the Hitmapp++ channel implementation, and the work-stealing mechanism.

Figure 5.16 shows the results of the Mandelbrot implementations. The programs calculate the set in a grid of $2^{14} \times 2^{13}$ elements. The programs use up to 1 000 iterations to determine if each element belongs to the set. FastFlow implementation scales better in Atlas, since the internal lock-free queues take advantage of the shared-memory architecture. Hitmap++ shows worse scalability in the shared-memory machine but obtains the same results as FastFlow in the distributed one. This shows that Hitmap++ channel and work-stealing implementation have a great scalability in distributed environments and there that is still room for improvement in shared memory machines.

Swps3

For the swps3 benchmark, we compare the original version [122], which is implemented using pipe and fork system calls to create several processes in the same machine, with the FastFlow and the Hitmap++ versions. The structure of this benchmark is a farm with an emitter. We have developed the FastFlow and Hitmap++ versions starting from the sequential code of the original swps3 benchmark. We have not used the original example included in FastFlow [6], since it uses some memory allocation optimizations and does not work for the bigger sequences chosen as input for our experiments, that allow to generate enough workload for our target systems. Thus, the results can be used to measure the performance of the frameworks fairly. All the versions match a single protein sequence with all the proteins from a database of sequences. We have used the UniProt Knowledgebase (UniProtKB) release 2014_04, a protein information database maintained by the Universal Protein Resource

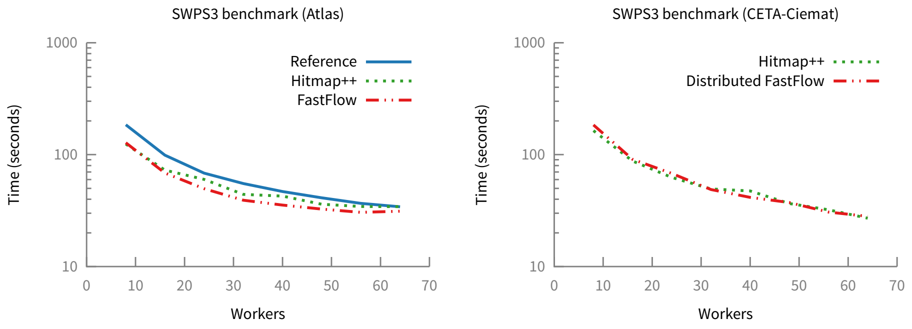


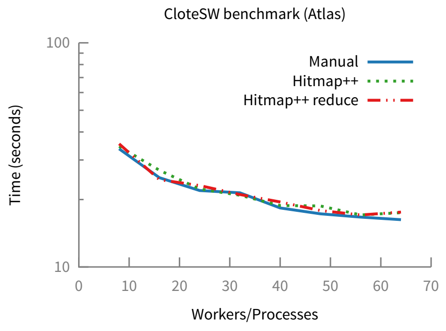
Figure 5.17: Swps3 benchmark results using the protein sequence Q8WXI7 as inputset.

(UniProt). This database consists of 544 996 sequences which minimum, maximum, and average lengths are 2, 35 213, and 355 respectively. Each sequence in the database is a task that will be fed to the farm, so they can be matched concurrently. With this example, we test the Hitmap++ communication and work-stealing performance on a real application.

Figure 5.17 shows the experimental results for the sequence named Q8WXI7, which has 22 152 proteins. For the shared-memory machine, both Hitmap++ and FastFlow surpass the results of the reference version. FastFlow obtains a slightly better performance than Hitmap++. In the cluster, there is no significant difference between both versions. We can conclude that Hitmap++ can be used for this kind of real applications with minimum performance degradation due to the proposed implementation.

Clote's algorithm

The third benchmark, CloteSW, is a different implementation of the Smith-Waterman protein alignment that aims to compare two big sequences [32]. For this benchmark, we compare two sequences of 100 000 elements. They are bigger than any of the sequences used in the previous experiment. For this case, the Smith-Waterman algorithm requires a 100 000 x 100 000 elements matrix to be calculated with the alignment. The computation is broken down into pieces, following a distributed wavefront structure. The benchmark has several phases: First, it populates the alignment matrix following the wavefront structure. Then, it performs a reduce operation to determine the maximum match sequence. Finally, it uses a backtracking method to compose the sequence traversing the wavefront structure in the inverse order. We have developed a manual C++ & MPI version, a Hitmap++ version, and another Hitmap++ version that uses the proposed all-to-all reduce places (see Sect. 5.4.5). The Hitmap++ implementations use the mode structure described in Fig. 5.5. The results of all versions are shown in Fig. 5.18. All the versions show a similar performance. The use of the modes in Hitmap++ allows the data affinity between phases of the benchmark to



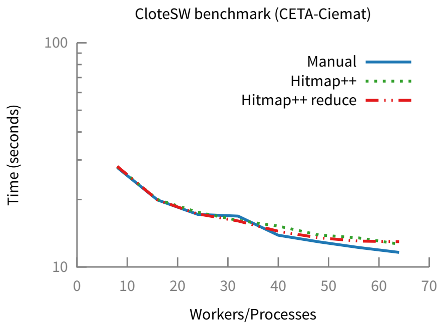


Figure 5.18: Clote's Smith-Waterman benchmark results.

be defined, avoiding extra communications or coordinations. Moreover, the use of reduce places simplifies the construction of the network.

Cellular Automata

The last tested benchmark is a cellular automata that performs 1 000 iterations of a 4-point stencil computation in a 10 000 x 10 000 bidimensional grid. We compare Hitmap++ against a manual version. The manual version is a classical stencil implementation that divides the grid into portions and has a distributed phase to update each grid partition in parallel using the edge contribution of other parts. The Hitmap++ version uses the same division, but each partition will be assigned to a transition that communicates with its neighbors, sending and receiving the edge values to update the grid using places in two different modes. The results in Fig. 5.19 show that Hitmap++ obtains a similar performance compared to the manual C+MPI version. These results show that the Hitmap++ model can also be applied to problems that are usually solved using static data mapping parallel models.

5.6.3 Code complexity

We use several code complexity and development-effort metrics to compare Hitmap++ code complexity against the other implementations. Tables 5.3 to 5.6 show the number of lines and tokens, the McCabe cyclomatic complexity [90], and the Halstead Mental discrimination [68], for the different considered versions of the benchmarks.

For the Mandelbrot set and the Swps3 Smith-Waterman benchmarks (Tables 5.3 and 5.4), all the metrics suggest that the regular FastFlow version and the Hitmap++ version are the simplest implementations. Both frameworks simplify the programming effort compared with the manual versions. However, the regular FastFlow version cannot be used in distributed memory systems. The extended distributed FastFlow version derives on bigger metrics values. This is due to the use of the two-tier model that forces the shared memory logic inside

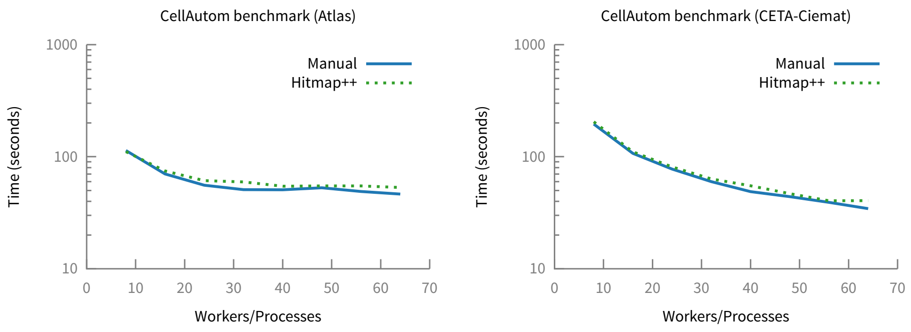


Figure 5.19: Cellular Automata benchmark results.

Metric	Manual MPI	FastFlow	Distributed FF	Hitmap++
Lines	237	169	326	189
Tokens	1497	1033	2272	1291
McCabe complexity	21	19	29	18
Halstead Mental discrimination	9.36×10^5	7.25×10^5	21.9×10^5	8.56×10^5

Table 5.3: Complexity comparison for the Mandelbrot set benchmark

Metric	Reference	FastFlow	Distributed FF	Hitmap++
Lines	68	40	48	28
Tokens	350	230	276	178
McCabe complexity	1039	672	834	481
Halstead Mental discrimination	1.77×10^6	7.24×10^5	1.03×10^6	4.15×10^5

Table 5.4: Complexity comparison for the Swps3 Smith-Waterman benchmark

Metric	Manual MPI	Hitmap++	Hitmap++ reduce places
Lines	512	787	740
Tokens	1385	2014	1915
McCabe complexity	79	141	126
Halstead Mental discrimination	3.27×10^6	8.57×10^6	7.15×10^6

Table 5.5: Complexity comparison for the Clote Smith-Waterman benchmark

Metric	Manual MPI	Hitmap++
Lines	220	270
Tokens	999	757
McCabe complexity	40	66
Halstead Mental discrimination	2.28×10^6	2.20×10^6

Table 5.6: Complexity comparison for the Cellular Automata benchmark

a node, and the distributed coordination logic to be implemented separately. This can be seen in Listing 5.6 and Listing 5.7, which show a full example of a simple pipeline application implemented in Hitmap++, FastFlow, and distributed FastFlow. Hitmap++ API is similar to FastFlow so the program implementation have nearly the same number of lines of code. The main difference is that Hitmap++ does not provide yet higher-level constructions, thus the activity network have to be created linking instances of places and transitions. Since Hitmap++ framework is designed to support both shared- and distributed-memory, the program in Listing 5.6 will work on a distributed environment by default. However, the regular FastFlow version cannot be used in distributed memory systems. The equivalent program using the extended distributed FastFlow version is shown in Listing 5.7.

For the Clote Smith-Waterman and the Cellular Automata benchmarks (Tables 5.5 and 5.6), Hitmap has bigger metric values than the manual version because these benchmarks have a static parallel structure, not based on dataflow parallelism.

5.7 Related work

Hitmap++ is a complement of Hitmap, a library for automatic but static hierarchical mapping, with support for dense and sparse data structures [52, 59]. Hitmap was previously described in Chapter 3. The Hitmap library focuses on data-parallel techniques and lacks support for dataflow applications. Our first approach, described in Sect. 5.2, was to introduce a dataflow model as a Hitmap extension. The second model introduced in this chapter (see Sect. 5.3) generalizes several restrictions of the previous one, introducing a complete generic model to represent any kind of combinations of parallel structures and paradigms.

Algorithm skeletons are a high-level parallel programming model that takes advantage of common programming patterns to hide the complexity of parallel programs. However, the skeleton libraries are limited by the actual set of parallel patterns that they offer. They can only be used when the program structure matches one of the provided skeletons. Thus, they cannot represent every kind of parallel application. Most of the skeleton libraries, such as eSkel [18] or Muesli [31], are intended for distributed environments. Some research is

```cpp
#include <hitmap++.h>
using namespace hitmap;

class StageA: public Transition {
   int numtasks;
public:
   StageA(int t): numtasks(t){};

   void create(){
      long task;
      for(int i=0; i<numtasks; i++){
         task = i;
         put(task);

      }

   }
};

class StageB: public Transition {
   long sum;
public:
   void init(){
      sum = 0;

   }
   void process(){
      long task;
      get(&task);
      sum += task;

   }
   void end(){
      cout << "Sum " << sum << endl;
   }
};

int main(int nargs, char * vargs[]){

   Hitmap::init(&nargs, &vargs);

   StageA st_a(10); StageB st_b;

   Place<long> place("long container");

   st_a.addOutput(&place,"createTasks");
   st_a.addMethod(&StageA::create,"createTasks");
   st_b.addMethod(&StageB::process,"processTasks");
   st_b.addInput(&place,"processTasks");

   Net net;
   net.add(&st_a); net.add(&st_b);
   net.run();

   return 0;
}
```

```cpp
#include <ff/pipeline.hpp>
using namespace ff;

class StageA: public ff_node {
   int numtasks;
public:
   StageA(int t): numtasks(t){};

   void * svc(void * intask){

      for(int i=0; i<numtasks; i++){
         long * task = new long;
         *task = (long) i;
         ff_send_out(task);
      }
      return NULL;
   }
};

class StageB: public ff_node {
   long sum;
public:
   int svc_init(){
      sum = 0;
      return 0;
   }
   void * svc(void * intask){
      long * task = (long*) intask;
      sum += *task;
      delete task;
      return GO_ON;
   }
   void svc_end(){
      cout << "Sum " << sum << endl;
   }
};

int main() {

   ff_pipeline pipe;
   pipe.add_stage(new StageA(10));
   pipe.add_stage(new StageB());
   if (pipe.run_and_wait_end()<0)
      return -1;
   return 0;
}
```

Listing 5.6: A full pipeline example in both Hitmap++ and FastFlow frameworks.

```
#include <ff/dnode.hpp>
using namespace ff;


char * address; // StageA address
zmqTransport*  trans; // ZeroMQ transport

class StageA: public ff_dnode<zmqOnDemand> {
   int numtasks;

public:
   StageA(int t): numtasks(t){};

   int svc_init() {
      ff_dnode<zmqOnDemand>::init("ChName",
      address, 1, trans, true, 0, callback);
      return 0;
    }

   void * svc(void * intask){
      for(int i=0; i<numtasks; i++){
         long * task = new long;
         *task = (long) i;
         ff_send_out(task);
      }
      return NULL;
   }

    void prepare(svector<iovec>& v,
      void* ptr, const int sender=-1) {

        struct iovec iov={ptr,sizeof(long)};
        v.push_back(iov);
    }

   static void callback(void * e,void* x) {
      long * l = (long*) e;
      delete l;
   }

};
class StageB: public ff_dnode<zmqOnDemand> {
   long sum;

public:

   int svc_init(){
      ff_dnode<zmqOnDemand>::init("ChName",
      address, 1, trans, false, 0);
      sum = 0;
      return 0;
   }

    virtual void unmarshalling(
      svector<msg_t*>* const v[],
      const int vlen, void *& task) {

      long * l = new long;
      msg_t * msg = v[0]->operator[](0);
      *l = *((long*)msg->getData());
        delete v[0];
      task = l;
    }

   void * svc(void * intask){
      long * task = (long*) intask;
      sum += *task;

      delete task;
      return GO_ON;
   }

   void svc_end(){
      cout << "Sum " << sum << endl;
   }
};

int main(int argc, char * argv[]) {

   // 0 StageA, 1 StageB
   int proc = atoi(argv[1]);
   address = argv[2];

   trans = new zmqTransport(proc);
   trans->initTransport();

   if(proc == 0){
      StageA * a = new StageA(10);
      a->run(); a->wait();
   } else {
      StageB * b = new StageB();
      b->run(); b->wait();
   }

   trans->closeTransport();
   return 0;
}
```

Listing 5.7: Pipeline example using the FastFlow distributed extension.

being done to enable parallel skeletons in different architectures. For example, a Muesli extension [30] targets heterogeneous systems. SkeTo [79] is a data parallel skeleton library for clusters of multi-core nodes. Hitmap++ does not yet provide the support of skeletons. However, high level of abstraction artifacts can be implemented using the Hitmap++ generic network elements. For the first model proposal (see Sect. 5.2.2), it has been shown that it can be used as building blocks for parallel skeletons implementation.

Hitmap++ has several similarities with FastFlow [5], a structured parallel programming framework targeting shared memory multi-core architectures. FastFlow is structured as a stack of layers that provide different levels of abstraction, providing the parallel programmer with a set of ready-to-use, parametric algorithmic skeletons, modeling the most common parallelism exploitation patterns. Hitmap++ API is similar to FastFlow. The main differences are that the Hitmap++ framework is designed to support both shared- and distributed-memory with a single tier model; that is, it includes a transparent mechanism for the correct termination of networks even in the presence of feedback-edges; and mode-driven control to create affinity between transitions in distributed memory environments. Regarding the system targeted, the FastFlow group is working on a distributed memory extension using a two-tier model [4]. However, this solution forces the programmer to manually divide the program structure for the available memory spaces and use a different mechanism of external channels to communicate the tasks. In this sense, Hitmap++ makes the program design independent from the mapping. Recall Listing 5.7, that shows the implementation of the previous two stage pipeline example. With Hitmap++, it is possible to create the network without knowing in advance what the chosen mapping will be. Thus, the code is portable to any distributed system. Currently, Hitmap++ does not provide higher-level constructions such as patterns or skeletons. However, our implementation is equivalent to the FastFlow low-level programming layer. We provide the transition and place entities to build MPMC structures. FastFlow provides low-level MPMC queues that are used by the patterns of the high-level programming layer, although without the transparent work-stealing mechanism in Hitmap++. Following the same methodology, it is possible to build a higher abstract layer implementing parallel patterns using Hitmap++ as building blocks. These higher-level constructions would inherit all the features of Hitmap++, supporting shared- and distributed-memory environments, and transparently including the load balancing mechanism.

OpenStream [103] is a dataflow OpenMP extension where dynamic independent tasks communicate through streams. The programmer exposes data flow information using pragmas to define the stream input and output task. This allows arbitrary dependence patterns between tasks to be created. A stream can have several producers and consumers that access the data in the stream using a sliding window. The OpenStream runtime ensures the coordination of the different elements. These streams are equivalent to the Hitmap++ places. The main differences between the two solutions are that OpenStream does not have any kind of load balancing mechanism, because the streams are a shared entity; and that OpenStream only focused on shared memory architectures. Trying to transfer this model to distributed memory involves the same problems as the use of distributed FastFlow.

S-Net [64, 65] is a declarative coordination language. It defines the structure of a program as a network of connected asynchronous components called boxes. S-Net only takes care of the coordination: The operations done inside boxes are defined using conventional languages. Boxes are stateless components with only a single input and a single output stream. Boxes can be combined hierarchically to create complex networks. S-Net defines four network combinations: serial and parallel composition, and serial and parallel replication. From the programmers' perspective, the implementation of streams on the language level by either shared memory buffers or distributed memory message passing is entirely transparent. Like Hitmap++, S-Net aims for shared and distributed support. However, its components are less flexible, since they do not support state, and it is not possible to define any kind of data affinity between components.

5.8 Conclusions

This chapter presents a new parallel programming model and framework based on the dataflow paradigm. This framework extends the Hitmap library allowing programs to be described as a network of communicating activities in an abstract form.

The main characteristics of this model can be summarized as: (1) The model is based on Colored Petri nets allowing typed places and signal tokens to be represented; (2) we present a general MPMC system where consumers can consume different task types from different producers; (3) it supports cycles in the network construction; (4) this model introduces a concept of mode inside the processing units to reconfigure the network, in order to allow mutual exclusion, and to intuitively define task-to-task affinity.

It can describe from simple static parallel structures to complex combinations of dataflow and dynamic parallel programs. The description is decoupled from the mapping techniques or policies, which can be efficiently applied at runtime, automatically adapting static or dynamic structures to different resource combinations. Our current framework transparently targets hybrid shared- and distributed-memory platforms.

With this extension Hitmap is now able to act as a runtime for a generic parallel programming system. Previous chapters showed that we can now generate automatic hierarchical mapping for static programs supporting dense and sparse data structures. In addition with these previous capabilities, Hitmap can be used now to develop generic dynamic programs using a dataflow approach.

Chapter 6

Conclusions

Nowadays, there is a huge increase in the capabilities and complexity of parallel platforms. To take advantage of these platforms, we require new programming tools. First, models with high-level abstractions to represent appropriately parallel algorithms. Second, the programming systems that implement these models need complete runtimes that offer different parallel paradigms to the system programmers to allow the building of programs that can be efficiently executed in different platforms. There are different areas to study in order to develop the ideal runtime system. This Ph.D. Thesis addresses two common problems, which where introduced in Chapter 1: The unified support for dense and sparse data, and the integration of data-mapping and data-flow parallelism. This chapter contains a summary of the Thesis contributions, the answer to the proposed research question, and it discusses possible future directions.

6.1 Summary of contributions

The Hitmap library has been used in this Thesis as a base to create a runtime system for a generic parallel programming system. This section summarizes the contributions of this Thesis and the related papers that have been published. The contributions include the previous work needed to study the automatic techniques of the Hitmap library and the main work done to solve the two identified problems that appear in current runtime systems: The unified support for dense and sparse data, and the integration of data-mapping and data-flow parallelism.

6.1.1 Previous work

Some previous work has been done in order to understand the capabilities of the original Hitmap library, and to determine which lacks should be solved. We had studied the automatic data-layout techniques of the library applied to the implementation of multigrid methods. We reviewed the multigrid methods, in particular the NAS MG benchmark [12]. For this benchmark, we developed a Hitmap implementation using the library plugin system to automatize the data layout and distribution. The work done led to the publication of the following papers:

1. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Automatic Data Partitioning Applied to Multigrid PDE Solvers'. In: *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*. February 9-11, Ayia Napa, Cyprus: IEEE, Feb. 2011, pp. 239–246. ISBN: 978-1-4244-9682-2. DOI: 10.1109/PDP.2011.38

2. 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

6.1.2 Unified support for dense and sparse data

One of the limitations of many parallel systems is that they do not provide support for both dense and sparse data structures. Our goal was to integrate dense and sparse support into the Hitmap library. To achieve it, we have first made an analysis of different data structures, algorithms, library tools, and programming frameworks to manage sparse data. Then, we have extended the parallel programming methodology of Hitmap to conceptually integrate dense and sparse domains. Finally, we have redesign the library to develop a new version following the proposed methodology. The new library implementation has several graph and sparse matrix representations that can be used with the same abstractions as the dense structures. These contributions have been published in the following papers:

3. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Integrating dense and sparse data partitioning'. In: *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*. ed. by J. Vigo-Aguiar. June 26-29, Alicante, Spain, 2011, pp. 532–543

4. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Extending a hierarchical tiling arrays library to support sparse data partitioning'. In: *The Journal of Supercomputing*, vol. 64, no. 1, 2013, pp. 59–68. Springer US, 2013. ISSN: 0920-8542. DOI: 10.1007/s11227-012-0757-y

5. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Data abstractions for portable parallel codes'. In: *Proceedings of the Ninth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. ed. by HiPEAC. 14-20 July, Fiuggi, Italy, 2013, pp. 85–88

6. 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

6.1.3 Integration of parallel paradigms and models with dynamic parallel structures

The second problem addressed in the Thesis is the support of paradigms and models with dynamic parallel structures. Simple static parallel structures are easy to program using common parallel programming tools. However, programming dynamic and dataflow applications is more challenging. Some available frameworks focus on a given parallel paradigm, or they can only be used for particular platforms. A ideal runtime for a generic programming system must support these challenging dynamic parallel paradigms and models, as well as classical static communication structures.

We have studied Petri nets [95] and Colored Petri nets [77] as formal modeling languages for the description of parallel systems. Based on these languages, we have developed a theoretical model for dataflow mechanisms. We have extended Hitmap to support the proposed model. This Hitmap version allow us to describe programs as reconfigurable networks of activities and typed data containers arbitrarily interconnected. The work done has been published in the following papers:

7. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Exploiting parallel skeletons in an all-purpose parallel programming system'. In: *Science and Supercomputing in Europe - research highlights (HPC-Europa2 project)*, 2012. 2012

8. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Runtime Support for Dynamic Skeletons Implementation'. In: *Proceedings of the 19th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. July 22-25, Las Vegas, NV, USA., 2013, pp. 320–326
9. Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. *Dataflow Programming Model for Hybrid Distributed and Shared Memory Systems*. Manuscript submitted for publication. 2015

6.2 Answer to the research question

This section answers the research question that was introduced in Chapter 1. The question was:

> *Is it possible to create a runtime system for a generic high level programming language that offers (1) common abstractions for dense and sparse data management, and (2) generic data-mapping and data-flow parallelism support for hybrid shared- and distributed-memory environments?*

The work presented in this Thesis allow us to conclude that the research question has an affirmative response. The Hitmap library was intended to be used as a runtime for the Trasgo parallel computing system. However, it lacked support for sparse data, and dynamic data-flow mechanism. With the research work done in this Thesis, we have been able to extend the library to remove these limitations.

6.3 Future work

We have focused on two particular problems in parallel computing at the level of the runtime tools. However, this work should be used in the context of building a complete parallel programing framework. This Thesis lets the door open to some questions that can be expanded upon in future research projects.

- *Transformation from high level code.* The Hitmap library is intended to be used as the runtime system of the Trasgo framework. The transformation from the Trasgo high level code to a lower code with Hitmap calls is an ongoing research work that should consider. In particular, for the new features introduced in this Thesis.
- *Mapping policies.* Hitmap allows to select a set of pre-defined mapping plug-ins and it also allow the programmer to create new ones. To give support to as many applications as possible, it is necessary to study new mapping policies in order to integrate them into the library. This is specially relevant for targeting heterogeneous platforms.

- *High level abstraction artifacts.* The dataflow model introduced for this Thesis can be used as a building blocks layer for higher abstractions. In particular, it is suitable to represent abstract constructions that hide the internal complex details of the parallel algorithm, such as algorithmic skeletons.

Bibliography

[1] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. Cambridge, Massachusetts. London, England: The MIT Press, 2002. ISBN: 0262011891.

[2] 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.

[3] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *Data structures and algorithms*. 1st ed. Boston, MA, USA: Addison Wesley, June 1983. ISBN: 0201000237.

[4] Marco Aldinucci, Sonia Campa and Marco Danelutto. 'Targeting distributed systems in fastflow'. In: *Proceedings of the Euro-Par 2012 Parallel Processing Workshops*. Vol. 7640. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 47–56. DOI: 10.1007/978-3-642-36949-0_7.

[5] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick and Massimo Torquati. 'FastFlow: high-level and efficient streaming on multi-core'. In: *Programming Multi-core and Many-core Computing Systems*. Ed. by Sabri Pllana and Fatos Xhafa. 1st. Parallel and Distributed Computing. Wiley, 2014. ISBN: 0470936908.

[6] Marco Aldinucci, Massimiliano Meneghin and Massimo Torquati. 'Efficient Smith-Waterman on Multi-core with FastFlow'. In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*. Ed. by Marco Danelutto, Julien Bourgeois and Tom Gross. Pisa, Italy: IEEE Computer Society, 2010, pp. 195–199. ISBN: 978-1-4244-5672-7. DOI: 10.1109/PDP.2010.93.

[7] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr. and Sam Tobin-Hochstadt. *The Fortress language specification*. Tech. rep. 2008. URL: http://www.eecis.udel.edu/~cavazos/cisc879-spring2008/papers/fortress.pdf.

[8] José Nelson Amaral, Renee Elio, Jim Hoover, Ioanis Nikolaidis, Mohammad Salavatipour, Lorna Stewart and Ken Wong. *About Computing Science Research Methodology*. 2007. URL: webdocs.cs.ualberta.ca/~c603/readings/research-methods.pdf.

[9] AndroidCentral. *HTC's first octa-core 64-bit smartphone is on the way*. URL: http://www.androidcentral.com/htcs-first-octa-core-64-bit-smartphone-way (visited on 09/02/2015).

[10] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti and Marco Vanneschi. 'P3L: A structured high-level parallel language, and its structured support'. In: *Concurrency Practice and Experience*, vol. 7, no. 3, 1995, pp. 225–255. John Wiley & Sons, Ltd, 1995. ISSN: 10403108. DOI: 10.1002/cpe.4330070305.

[11] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. *The NAS Parallel Benchmarks*. Report RNR-94-007. 1994.

[12] 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.

[13] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith and Hong Zhang. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.5. 2014. URL: http://www.mcs.anl.gov/petsc.

[14] Satish Balay, William D. Gropp, Lois Curfman McInnes and Barry F. Smith. 'Efficient Management of Parallelism in Object Oriented Numerical Software Libraries'. In: *Modern Software Tools in Scientific Computing*. Ed. by Erlend Arge, Are Magnus Bruaset and Hans Petter Langtangen. Birkhäuser Boston, 1997. Chap. 8, pp. 163–202.

[15] Blaise Barney. *POSIX Threads Programming*. Lawrence Livermore National Laboratory, 2014. URL: https://computing.llnl.gov/tutorials/pthreads (visited on 07/09/2014).

[16] Richard Barrett, Michael Berry, Tony. F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2º. Vol. 64. 211. SIAM, July 1994. DOI: 10.2307/2153507.

[17] Anne Benoit and Murray Cole. 'Two Fundamental Concepts in Skeletal Parallel Programming'. In: *5th International Conference on Computational Science (ICCS 2005)*. Ed. by Vaidy S. Sunderam, G. Dick van Albada, Peter M. A. Sloot and Jack Dongarra. Atlanta, GA, USA: Springer, 2005, pp. 764–771. DOI: 10.1007/11428848_98.

[18] Anne Benoit, Murray Cole, Stephen Gilmore and Jane Hillston. 'Flexible skeletal programming with eSkel'. In: *Proceedings of the 11th international Euro-Par conference on Parallel Processing*. Lisbon, Portugal: Springer-Verlag, 2005, pp. 761–770. DOI: 10.1007/11549468_83.

[19] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almási, Basilio B. Fraguela, David Padua, Christoph von Praun and María J. Garzarán. 'Programming for parallelism and locality with hierarchically tiled arrays'. In: *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*. New York, New York, USA: ACM Press, 2006, pp. 48–57. ISBN: 1595931899. DOI: 10.1145/1122971.1122981.

[20] Ronald F. Boisvert, Roldan Pozo and Karin A. Remington. *The Matrix Market Exchange Formats: Initial Design*. Tech. rep. Gaithersburg, MD, USA, 1996. URL: math.nist.gov/MatrixMarket/reports/MMformat.ps.gz.

[21] Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Amir Kamil, Ben Liblit, Geoff Pike, Jimmy Su and Katherine A. Yelick. *Titanium Language Reference Manual. Version 2.20*. Tech. rep. Berkely, CA, USA, 2006. URL: titanium.cs.berkeley.edu/doc/lang-ref.pdf.

[22] BrightHand. *LG Announces Smartphone with Dual-Core Processor*. URL: http://www.brighthand.com/news/lg-announces-smartphone-with-dual-core-processor (visited on 09/02/2015).

[23] William W. Carlson, Jesse M. Draper, David E. Culler, Katherine A. Yelick, Eugene Brooks and Karen Warren. *Introduction to UPC and Language Specification*. Tech. rep. CCS-TR-99-157. 1999. URL: www.gwu.edu/~upc/publications/upctr.pdf.

[24] Denis Caromel, Christian Delbé, Alexandre di Costanzo and Mario Leyton. 'ProActive: an integrated platform for programming and running applications on grids and P2P systems'. In: *Computational Methods in Science and Technology*, vol. 12, no. 1, 2006, pp. 69–77. OWN, Poznan Supercomputing and Networking Center, 2006. DOI: 10.12921/cmst.2006.12.01.69-77.

[25] Denis Caromel and Mario Leyton. 'Fine tuning algorithmic skeletons'. In: *Proceedings of the 13th International Euro-Par Conference*. Lecture Notes in Computer Science. Rennes, France: Springer Berlin Heidelberg, 2007, pp. 72–81. DOI: 10.1007/978-3-540-74466-5_9.

[26] Bradford L. Chamberlain. 'A brief Overview of Chapel'. In: *(Pre-print of an upcomming book chapter)*. 2013. Chap. 9.

[27] Bradford L. Chamberlain, Steven J. Deitz, David Iten and Sung-Eun Choi. 'User-defined distributions and layouts in chapel: philosophy and framework'. In: *Proceedings of the second USENIX conference on Hot topics in parallelism*. HotPar'10. June 14-15, Berkeley, CA, USA: USENIX Association, 2010, p. 12.

[28] 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.

[29] Barbara Chapman, Gabriele Jost and Rudd van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific and Engineering Computation Series. Cambridge, MA: MIT Press, 2008.

[30] Philipp Ciechanowicz and Herbert Kuchen. 'Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures'. In: *Proceedings of the IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*. Melbourne, VIC: IEEE, Sept. 2010, pp. 108–113. ISBN: 978-1-4244-8335-8. DOI: 10.1109/HPCC.2010.23.

[31] Philipp Ciechanowicz, Michael Poldner and Herbert Kuchen. *The Münster Skeleton Library Muesli – A Comprehensive Overview*. Tech. rep. 2009.

[32] Peter Clote. *Biologically significant sequence alignments using Boltzmann probabilities*. Tech. rep. 2003. URL: http://bioinformatics.bc.edu/~clote/pub/boltzmannParis03.pdf.

[33] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. ISBN: 0-262-53086-4.

[34] Murray Cole. 'Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming'. In: *Parallel Computing*, vol. 30, no. 3, Mar. 2004, pp. 389–406. Mar. 2004. ISSN: 01678191. DOI: 10.1016/j.parco.2003.12.002.

[35] Intel Corporation. *Intel ARK (Automated Relational Knowledgebase)*. URL: http://ark.intel.com (visited on 04/02/2015).

[36] Nvidia Corporation. *CUDA Parallel Computing Platform site*. URL: http://www.nvidia.com/object/cuda_home_new.html (visited on 05/02/2015).

[37] Nvidia Corporation. *GPU Supercomputers show exponential growth in Top500 list*. URL: http://blogs.nvidia.com/blog/2011/11/14/gpu-supercomputers-show-exponential-growth-in-top500-list (visited on 05/01/2015).

[38] Timothy A. Davis and Yifan Hu. 'The University of Florida Sparse Matrix Collection'. In: *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 2011, 1:1–1:25. 2011. URL: http://www.cise.ufl.edu/research/sparse/matrices.

[39] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: Simplified data processing on large clusters'. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*. San Francisco, CA, USA: USENIX Association, 2004, pp. 137–149.

[40] Jack Dongarra and Mark Gates. *Freely Available Software for Linear Algebra (May 2013)*. 2013. URL: http://www.netlib.org/utk/people/JackDongarra/la-sw.html.

[41] Jack Dongarra, Robert Graybill, William Harrod and Robert Lucas. 'DARPA's HPCS program: History, models, tools, languages'. In: *Advances in Computers*, vol. 72, 2008, pp. 1–94. Elsevier Ltd., 2008. DOI: 10.1016/S0065-2458(08)00001-6.

[42] Iain S. Duff, Roger G. Grimes and John G. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release 1)*. Tech. rep. 1992.

[43] Per-Olof Fjällström. 'Algorithms for Graph Partitioning: A Survey'. In: *Linköping Electronic Articles in Computer and Information Science*, vol. 3, no. 10, 1998. Linköping University Electronic Press, 1998. ISSN: 1401-9841. URL: http://www.ep.liu.se/ea/cis/1998/010/.

[44] Basilio B. Fraguela, Ganesh Bikshandi, Jia Guo, María J. Garzarán, David Padua and Christoph von Praun. 'Optimization techniques for efficient HTA programs'. In: *Parallel Computing*, vol. 38, no. 9, Sept. 2012, pp. 465–484. Elsevier B.V., Sept. 2012. ISSN: 01678191. DOI: 10.1016/j.parco.2012.05.002.

[45] Basilio B. Fraguela, Jia Guo, Ganesh Bikshandi, María J. Garzarán, Gheorghe Almási, José Moreira and David Padua. 'The Hierarchically Tiled Arrays programming approach'. In: *Proceedings of the 7th workshop on Workshop on Languages, Compilers, and Run-time support for scalable systems (LCR)*. Houston, TX, USA: ACM Press, 2004, pp. 1–12. DOI: 10.1145/1066650.1066657.

[46] Ricardo Freitas. 'Scientific Research Methods and Computer Science'. In: *Proceedings of the MAP-I Seminars Workshop*. Porto, Portugal, 2009.

[47] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Automatic Data Partitioning Applied to Multigrid PDE Solvers'. In: *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*. February 9-11, Ayia Napa, Cyprus: IEEE, Feb. 2011, pp. 239–246. ISBN: 978-1-4244-9682-2. DOI: 10.1109/PDP.2011.38.

[48] 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.

[49] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Data abstractions for portable parallel codes'. In: *Proceedings of the Ninth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Ed. by HiPEAC. 14-20 July, Fiuggi, Italy, 2013, pp. 85–88.

[50] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. *Dataflow Programming Model for Hybrid Distributed and Shared Memory Systems*. Manuscript submitted for publication. 2015.

[51] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Exploiting parallel skeletons in an all-purpose parallel programming system'. In: *Science and Supercomputing in Europe - research highlights (HPC-Europa2 project)*, 2012. 2012.

[52] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Extending a hierarchical tiling arrays library to support sparse data partitioning'. In: *The Journal of Supercomputing*, vol. 64, no. 1, 2013, pp. 59–68. Springer US, 2013. ISSN: 0920-8542. DOI: 10.1007/s11227-012-0757-y.

[53] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Integrating dense and sparse data partitioning'. In: *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*. Ed. by J. Vigo-Aguiar. June 26-29, Alicante, Spain, 2011, pp. 532–543.

[54] Javier Fresno, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Runtime Support for Dynamic Skeletons Implementation'. In: *Proceedings of the 19th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. July 22-25, Las Vegas, NV, USA., 2013, pp. 320–326.

[55] Mehdi Goli, John McCall, Christopher Brown, Vladimir Janjic and Kevin Hammond. 'Mapping parallel programs to heterogeneous CPU/GPU architectures using a Monte Carlo Tree Search'. In: *IEEE Congress on Evolutionary Computation*. Cancún, México: IEEE, June 2013, pp. 2932–2939. ISBN: 978-1-4799-0454-9. DOI: 10.1109/CEC.2013.6557926.

[56] Gene H. Golub and Charles F. Loan Van. *Matrix computations*. 3rd. The Johns Hopkins University Press, 1996.

[57] Jorge González-Domínguez, Óscar García-López, Guillermo L. Taboada, María J. Martín and Juan Touriño. 'Performance evaluation of sparse matrix products in UPC'. In: *The Journal of Supercomputing*, vol. 64, no. 1, 2013, pp. 100–109. 2013.

[58] Arturo Gonzalez-Escribano and Diego R. Llanos. 'Trasgo: a nested-parallel programming system'. In: *The Journal of Supercomputing*, Dec. 2009. Dec. 2009. ISSN: 0920-8542. DOI: 10.1007/s11227-009-0367-5.

[59] 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.

[60] Horacio González-Vélez and Mario Leyton. 'A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers'. In: *Software: Practice and Experience*, vol. 40, no. 12, 2010, pp. 1135–1160. Wiley Online Library, 2010. DOI: 10.1002/spe.1026.

[61] Sergei Gorlatch and Murray Cole. 'Parallel Skeletons'. In: *Encyclopedia of Parallel Computing*. Ed. by David A. Padua. Springer, 2011. ISBN: 978-0-387-09765-7.

[62] Ananth Grama, Anshul Gupta, George Karypis and Vipin Kumar. *Introduction to Parallel Computing*. 2º. Addison Wesley, 2003, p. 856. ISBN: 0201648652.

[63] Clemens Grelck, Jukka Julku and Frank Penczek. 'Distributed S-Net: Cluster and Grid Computing without the Hassle'. In: *Cluster, Cloud and Grid Computing (CCGrid'12), 12th IEEE/ACM International Conference, Ottawa, Canada*. IEEE Computer Society, 2012.

[64] Clemens Grelck, Sven-Bodo Scholz and Alex Shafarenko. 'A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components'. In: *Parallel Processing Letters*, vol. 18, no. 2, 2008, pp. 221–237. 2008.

[65] Clemens Grelck, Sven-Bodo Scholz and Alex Shafarenko. 'Asynchronous Stream Processing with S-Net'. In: *International Journal of Parallel Programming*, vol. 38, no. 1, 2010, pp. 38–67. 2010. DOI: 10.1007/s10766-009-0121-x.

[66] 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.

[67] Trasgo Research Group. *Trasgo webpage*. URL: http://trasgo.infor.uva.es (visited on 19/05/2015).

[68] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[69] Bruce Hendrickson and Robert Leland. 'A multilevel algorithm for partitioning graphs'. In: *Proceedings of the IEEE/ACM Supercomputing Conference*. New York, New York, USA: ACM Press, 1995. ISBN: 0897918169. DOI: 10.1145/224170.224228.

[70] Bruce Hendrickson and Robert Leland. *The Chaco User's Guide: Version 2.0*. Tech. rep. Albuquerque, NM, 1995.

[71] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Tech. rep. 1993.

[72] Alex Holmes. *Hadoop in Practice*. Greenwich, CT, USA: Manning Publications Co., 2012. ISBN: 1617290238, 9781617290237.

[73] David V. Hutton. *Fundamentals of finite element analysis*. McGraw-Hill, 2004. ISBN: 9780071122313.

[74] Google Inc. *Google Scholar*. URL: https://scholar.google.es (visited on 10/02/2015).

[75] Antonio J. Dios, Angeles Navarro, Rafael Asenjo, Francisco Corbera and Emilio L. Zapata. 'A case study of the task-based parallel wavefront pattern'. In: *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*. Melbourne, Australia, September 1-3, 2010.

[76] Kurt Jensen and Lars M. Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer, 2009. ISBN: 978-3-642-00283-0.

[77] Kurt Jensen, Lars Michael Kristensen and Lisa Wells. 'Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems'. In: *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3-4, Mar. 2007, pp. 213–254. Springer-Verlag, Mar. 2007. ISSN: 1433-2779. DOI: 10.1007/s10009-007-0038-x.

[78] JTC1/SC22/WG5. *Fortran 2008 standard (ISO/IEC 2010)*. Tech. rep. J3/10-007. 2010.

[79] Yuki Karasawa and Hideya Iwasaki. 'A Parallel Skeleton Library for Multi-core Clusters'. In: *Proceeding of the International Conference on Parallel Processing*. Vienna, Austria: IEEE, Sept. 2009, pp. 84–91. ISBN: 978-1-4244-4961-3. DOI: 10.1109/ICPP.2009.18.

[80] George Karypis and Vipin Kumar. *MeTiS–A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices–Version 4.0.* Tech. rep. 1998. URL: http://glaros.dtc.umn.edu/gkhome/views/metis.

[81] George Karypis and Vipin Kumar. 'Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs'. In: *SIAM Review*, vol. 41, no. 2, 1999, pp. 278–300. SIAM, 1999. ISSN: 0036-1445. DOI: 10.1137/S0036144598334138.

[82] Ken Kennedy, Charles Koelbel and Hans Zima. 'The rise and fall of High Performance Fortran'. In: *Proceedings of the thrid ACM SIGPLAN conference on History of programming languages (HOPL III)*. Vol. 54. 11. New York, NY, USA: ACM Press, 2007, pp. 7.1–7.22. DOI: 10.1145/1238844.1238851.

[83] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd ed. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[84] David B. Kirk and Wen-mei W. Hwu. *In Praise of Programming Massively Parallel Processors : A Hands-on Approach*. 1st ed. Morgan Kaufmann, 2010. ISBN: 9780123814722.

[85] Ralf Lämmel. 'Google's MapReduce programming model — Revisited'. In: *Science of Computer Programming*, vol. 70, no. 1, Jan. 2008, pp. 1–30. Elsevier, Jan. 2008. ISSN: 01676423. DOI: 10.1016/j.scico.2007.07.001.

[86] Mario Leyton and José M. Piquer. 'Skandium: Multi-core Programming with Algorithmic Skeletons'. In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb. 2010, pp. 289–296. IEEE, Feb. 2010. DOI: 10.1109/PDP.2010.26.

[87] Qingda Lu, Jiesheng Wu, Dhabaleswar Panda and P. Sadayappan. 'Applying MPI derived datatypes to the NAS benchmarks: A case study'. In: *Proceedings of the International Conference on Parallel Processing Workshops*. Montreal, QC, Canada: IEEE, 2004, pp. 538–545. ISBN: 0769521983. DOI: 10.1109/ICPPW.2004.1328066.

[88] Benoit B. Mandelbrot. 'Fractal aspects of the iteration of $z \mapsto \lambda z(1-z)$'. In: *Annals of the New York Academy of Sciences*, vol. 357, 1980, pp. 249–259. Wiley, 1980. DOI: 10.1111/j.1749-6632.1980.tb29690.x.

[89] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto and Zhenjiang Hu. 'A library of constructive skeletons for sequential style of parallel programming'. In: *Proceedings of the first international conference on Scalable information systems (InfoScale)*, 2006. New York, New York, USA: ACM Press, 2006. DOI: 10.1145/1146847.1146860.

[90] Thomas J. McCabe. 'A complexity measure'. In: *IEEE Transactions on software Engineering*, vol. SE-2, no. 4, 1976, pp. 308–320. IEEE, 1976. DOI: 10.1109/TSE.1976.233837.

[91] Larry Meadows. *The Birth of OpenMP 3.0.* Tech. rep. 2007. URL: http://openmp.org/wp/presos/omp30.pdf.

[92] John Mellor-Crummey, Laksono Adhianto, William N. Scherer and Guohua Jin. 'A new vision for coarray Fortran'. In: *Proceedings of the thrid Conference on Partitioned Global Address Space Programing Models (PGAS)*, 2009. New York, New York, USA: ACM Press, 2009. DOI: 10.1145/1809961.1809969.

[93] 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.

[94] Burkhard Monien and Stefan Schamberger. 'Graph Partitioning with the Party Library: Helpful-Sets in Practice'. In: *Proceedings of the16th Symposium on Computer Architecture and High Performance Computing*. IEEE, 2004, pp. 198–205. ISBN: 0-7695-2240-8. DOI: 10.1109/SBAC-PAD.2004.18.

[95] Tadao Murata. 'Petri nets: Properties, analysis and applications'. In: *Proceedings of the IEEE*, vol. 77, no. 4, 1989, pp. 541–580. 1989. DOI: 10.1109/5.24143.

[96] Sadegh Nobari, Xuesong Lu, Panagiotis Karras and Stéphane Bressan. 'Fast random graph generation'. In: *Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT)*. New York, New York, USA: ACM Press, 2011, pp. 331–342. ISBN: 9781450305280. DOI: 10.1145/1951365.1951406.

[97] OpenMP Architecture Review Board. *OpenMP Application Program Interface (Version 3.0)*. Tech. rep. 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[98] François Pellegrini. *PT-Scotch and libScotch 5.1 User's Guide*. Tech. rep. 2010. URL: http://gforge.inria.fr/docman/view.php/248/7104/scotch_user5.1.pdf.

[99] PhoneArena. *LG Optimus 4X HD detailed - the world's first quad-core phone is slim and powerful*. URL: http://www.phonearena.com/news/LG-Optimus-4X-HD-detailed---the-worlds-first-quad-core-phone-is-slim-and-powerful_id27167 (visited on 09/02/2015).

[100] Michael Poldner and Herbert Kuchen. 'Algorithmic Skeletons for Branch and Bound'. In: *Software and Data Technologies*, vol. 10, 2008, pp. 204–219. Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-70621-2_17.

[101] Michael Poldner and Herbert Kuchen. 'On Implementing the Farm Skeleton'. In: *Parallel Processing Letters*, vol. 18, no. 1, 2008, pp. 117–131. World Scientific, 2008. ISSN: 0129-6264. DOI: 10.1142/S0129626408003260.

[102] Michael Poldner and Herbert Kuchen. 'Task Parallel Skeletons for Divide and Conquer'. In: *Proceedings of Workshop of the Working Group Programming Languages and Computing Concepts of the German Computer Science Association GI*. Bad Honnef, Germany, 2008.

[103] Antoniu Pop and Albert Cohen. 'A OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs'. In: *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, 2013, 53:1–53:25. 2013. DOI: 10.1145/2400682.2400712.

[104] Nicolae Popovici and Thomas Willhalm. 'Putting intel® threading building blocks to work'. In: *Proceedings of the first International Workshop on Multicore Software Engineering (IWMSE)*, 2008, pp. 3–4. New York, New York, USA: ACM Press, 2008. DOI: 10.1145/1370082.1370085.

[105] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. *Numerical Recipes in C*. 2nd ed. Cambridge University Press, 1992. ISBN: 0-521-43108-5.

[106] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems*. 3rd ed. Berlin, Heidelberg: Springer, 2010. ISBN: 978-3-642-04817-3. DOI: 10.1007/978-3-642-04818-0.

[107] James Reinders. *Intel® threading building blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 1st ed. O'Reilly Media, 2007. ISBN: 978-0-596-51480-8.

[108] Torbjørn Rognes and Erling Seeberg. 'Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors'. In: *Bioinformatics*, vol. 16, no. 8, Aug. 2000, pp. 699–706. Oxford University Press, Aug. 2000. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/16.8.699.

[109] Youcef Saad. *SPARSKIT: a basic tool kit for sparse matrix computations*. Tech. rep. 1994. URL: http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps.

[110] Vijay A. Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu and David Grove. *X10 language specification (Version 2.5)*. Tech. rep. 2014.

[111] Vijay A. Saraswat, David Cunningham, Olivier Tardieu, Mikio Takeuchi, David Grove and Benjamin Herta. *A Brief Introduction To X10 (For the High Performance Programmer)*. Tech. rep. 2012. URL: http://x10.sourceforge.net/documentation/intro/intro-223.pdf.

[112] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky and Olivier Tardieu. 'The Asynchronous Partitioned Global Address Space Model'. In: *Proceedings of the first Workshop on Advances in Message Passing (co-located with PLDI 2010)*. Toronto, Canada: ACM, 2010.

[113] A Sarje, J Zola and S Aluru. *Scientific Computing with Multicore and Accelerators*. Ed. by Kurzak J, D A Bader and J Dongarra. Chapman and Hall/CRC, 2010. ISBN: 9781439825365.

[114] Palash Sarkar. 'A brief history of cellular automata'. In: *ACM Computing Surveys*, vol. 32, no. 1, Mar. 2000, pp. 80–107. Mar. 2000. ISSN: 03600300. DOI: 10.1145/349194.349202.

[115] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. 3rd ed. Dover Publications, 2010. ISBN: 048648582X.

[116] Temple F. Smith and Michael S. Waterman. 'Identification of common molecular subsequences'. In: *Journal of Molecular Biology*, vol. 147, no. 1, 1981, pp. 195–197. Elsevier B.V., 1981. DOI: 10.1016/0022-2836(81)90087-5.

[117] Matthew Sottile. 'Cellular Automata'. In: *Encyclopedia of Parallel Computing*. Ed. by David A. Padua. Springer, 2011. ISBN: 978-0-387-09765-7.

[118] Ivan P. Stanimirović and Milan B. Tasić. 'Performance comparison of storage formats for sparse matrices'. In: *Facta universitatis*. Mathematics and Informatics, vol. 24, no. 1, 2009, pp. 39–51. University of Niš, Serbia, 2009.

[119] Pyrros Theofanis Stathis. 'Sparse Matrix Vector Processing Formats'. PhD thesis. Delft University of Technology, 2004. ISBN: 90-9018828-2. URL: http://ce-publications.et.tudelft.nl/publications/969_sparse_matrix_vector_processing_formats.pdf.

[120] Michel Steuwer, Philipp Kegel and Sergei Gorlatch. 'SkelCL – A Portable Skeleton Library for High-Level GPU Programming'. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. Anchorage, AK, USA: IEEE, 2011, pp. 1176–1182. DOI: 10.1109/IPDPS.2011.269.

[121] Erich Strohmaier, Erich Dongarra, Horst Simon and Martin Meuer. *Top500 Supercomputer site*. URL: http://www.top500.org (visited on 04/02/2015).

[122] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl and Christophe Dessimoz. 'SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2.' In: *BMC research notes*, vol. 1, no. 107, 2008. BioMed Central Ltd., 2008. ISSN: 1756-0500. DOI: 10.1186/1756-0500-1-107.

[123] Yuri Torres. 'Hierarchical Transparent Programming for Heterogeneous Computing'. PhD thesis. Universidad de Valladolid, 2014. URL: http://www.infor.uva.es/~yuri.torres/docs/torres14.pdf.

[124] Yuri Torres, Arturo Gonzalez-Escribano and Diego R. Llanos. 'Encapsulated synchronization and load-balance in heterogeneous programming'. In: *Euro-Par Parallel Processing*. Vol. 7484. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 502–513. ISBN: 9783642328190. DOI: 10.1007/978-3-642-32820-6_50.

[125] UPC Consortium. *UPC Language Specifications Version 1.3*. Tech. rep. 2013. URL: https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf.

[126] Anila Usman, Mikel Luján, Len Freeman and John Gurd. 'Performance Evaluation of Storage Formats for Sparse Matrices in Fortran'. In: *High Performance Computing and Communications*. Lecture Notes in Computer Science, vol. 4208, 2006, pp. 160–169. Springer Berlin Heidelberg, 2006. DOI: 10.1007/11847366_17.

[127] Richard Vudac, James W. Demmel and Katherine A. Yelick. *The Optimized Sparse Kernel Interface (OSKI) Library*. Tech. rep. 2007. URL: http://bebop.cs.berkeley.edu/oski/oski-ug.pdf.

[128] Richard Vuduc, James W. Demmel and Katherine A. Yelick. 'OSKI: A library of automatically tuned sparse matrix kernels'. In: *Proceedings of Scientific Discovery through Advanced Computing (SciDAC)*. Journal of Physics: Conference Series. San Francisco, CA, USA: Institute of Physics Publishing, June 2005.

[129] Chris Walshaw. *The serial JOSTLE library user guide: Version 3.0*. Tech. rep. 2002. URL: http://staffweb.cms.gre.ac.uk/~wc06/jostle/jostleslib.pdf.

[130] Chris Walshaw and Mark Cross. 'Jostle: parallel multilevel graph-partitioning software - an overview'. In: *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Saxe-Coburg Publications on Computational Engineering. Civil-Comp Ltd, 2007, pp. 27–58. ISBN: 978-1-874672-29-6.

[131] Agnieszka Wegrzyn, Andrei Karatkevich and Jacek Bieganowski. 'Detection of deadlocks and traps in Petri nets by means of Thelen's prime implicant method'. In: *International Journal of Applied Mathematics and Computer Science*, vol. 14, no. 1, 2004, pp. 113–121. University of Zielona Gora Press, 2004.

[132] Wikipedia. *Personal computer — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/wiki/Personal_computer (visited on 03/02/2015).

[133] Wikipedia. *Smartphone — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/wiki/Smartphone (visited on 09/02/2015).

[134] Wikipedia. *Supercomputer — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/wiki/Supercomputer (visited on 03/02/2015).

[135] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2004, p. 496. ISBN: 0131405632.

[136] Michael M. Wolf, Erik G. Boman and Bruce A. Hendrickson. 'Optimizing parallel sparse matrix-vector multiplication by corner partitioning'. In: *Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*. Trondheim, Norway: Springer, 2008.

[137] Katherine Yelick, Susan L. Graham, Paul Hilfinger, Dan Bonachea, Jimmy Su, Amir Kamil, Kaushik Datta, Phillip Colella and Tong Wen. 'Titanium'. In: *Encyclopedia of Parallel Computing*. Ed. by David A. Padua. Springer, 2011. ISBN: 978-0-387-09765-7.

[138] Raphael Yuster and Uri Zwick. 'Fast sparse matrix multiplication'. In: *ACM Transactions on Algorithms*, vol. 1, no. 1, 2005, pp. 2–13. ACM, 2005. ISSN: 15496325. DOI: 10.1145/1077464.1077466.