An extensible system for multilevel automatic data partition and mapping

Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos, *Senior Member, IEEE*

Abstract—Automatic data distribution is a key feature to obtain efficient implementations from abstract and portable parallel codes. We present a highly efficient and extensible runtime library that integrates techniques for automatic data partition and mapping. It uses a novel approach to define an abstract interface and a plug-in system to encapsulate different types of regular and irregular techniques, helping to generate codes which are independent of the exact mapping functions selected. Currently, it supports hierarchical tiling of arrays with dense and stride domains, that allows the implementation of both data and task parallelism using a SPMD model. It automatically computes appropriate domain partitions for a selected virtual topology, mapping them to available processors with static or dynamic load-balancing techniques. Our library also allows the construction of reusable communication patterns that efficiently exploit MPI communication capabilities. The use of our library greatly reduces the complexity of data distribution and communication, hiding the details of the underlying architecture. The library can be used as an abstract layer for building generic tiling operations as well. Our experimental results show that the use of this library allows to achieve similar performance as carefully-implemented manual versions for several, well-known parallel kernels and benchmarks in distributed and multicore systems, and substantially reduces programming effort.

Index Terms—Data partition, mapping techniques, tiling, parallel libraries, MPI.

1 Introduction

Tiling is a well-known technique used to distribute data and tasks in parallel programs [1] and to improve the locality of loop nests in parallel and sequential code [2]. Although originally presented as a loop transformation technique, the use of data structures to support tiles for generic arrays allows to better exploit the memory hierarchy, since data is often reused within a tile. Tiling can be applied to multiple levels, to distribute work among processors at the outermost level, while locality are enhanced at the innermost level. In the context of distributed-memory, tiles can also make communication explicit, since computations involving elements from different tiles result in data movement [3], [4].

During the last decade, different programming models have been proposed to handle the complexity of multi-level data partition and mapping. These programming models roughly falls into two categories: Those that hidden the underlying communications (e.g. Chapel [5], UPC [6]), and those where the explicit communication is driven by the partition made by the user (e.g. MPI). These parallel programming models do not help the programmer to explicitly express the communication pattern needed by the algorithm regardless of the data partition chosen. Parallel programming tools and frameworks presented in the last years do not establish clear boundaries between virtual topologies, layouts as domain partitions, and tile management (such as HTA [7] or UPC), or clear boundaries between data management and communications (such as Chapel, UPC or HTA). Such a division of duties would allow the programmer to decouple the communication structures, that depend on the algorithm characteristics, from the data partition mechanisms.

In this context we have developed Hitmap, a library designed to decouple the communication pattern from data partitioning, thanks to the use of abstract expressions of the communications that are automatically adapted at runtime depending on the partition finally used. Hitmap presents an unique combination of features, including: (1) An extensible plug-in system, based on two different types of modules, to automatically compute data-partition and distributions of tiles as a function of the topology of the underlying architecture, hiding the details to the programmer; (2) a common framework to program new plug-ins with regular, irregular, static, or dynamic partitioning and load balancing techniques; (3) a flexible toolset for data- and task-parallelism mapping with a common interface; (4) an API to create complex and scalable communication patterns in terms of an abstract partition and layout. All these features allow to embed complex mapping decisions, some of them associated to compiler technology, in a library. These features make Hitmap an excellent choice to develop higher-level programming models [8].

Hitmap can be used to support complex data structures, such as sparse matrices and graphs for irregular applications [9], using the same hierarchical tiling methodology, or to program heterogeneous system [10]. The result is a good balance between performance and efficient memory usage, also reducing the programming

• *Departamento de Informática, ETS de Ing. Informática, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain. E-mail: {arturo,yuri.torres,jfresno,diego}@infor.uva.es.*

effort compared with other options.

Hitmap is designed with an object-oriented approach, internally exploiting several efficient MPI techniques for communication, focusing on performance and on further native compiler optimizations. Thus, the implementation is highly efficient, as we show with experimental results for well-known parallel benchmarks.

The rest of the paper is organized as follows. Section 2 describes the Hitmap library and how to use it to implement applications and kernels. Section 3 describes Hitmap architecture details, including tiling management, data partition plug-ins, and communications. Section 4 shows experimental results. Section 5 discusses other approaches for automatic data partition and mapping. Finally, Section 6 concludes this paper.

2 The Hitmap library

In this section we describe the library, starting with several key concepts and notations about arrays and tiles.

Signatures: We define a *Signature* S as a tuple of three integer elements representing a subspace of array indexes in a one-dimensional domain. It resembles the classical Fortran90 or MATLAB notation for array-index selections. The cardinality of the signature is the number of different indexes in the domain.

$$S \in \text{Signature} = (\text{begin} : \text{end} : \text{stride})$$
$$\text{Card}(s \in \text{Signature}) = \lfloor (\text{s.end} - \text{s.begin})/\text{s.stride} \rfloor$$

Shapes: We define a *Shape* h as a n-tuple of signatures. It represents a selection of a subspace of array indexes in a multidimensional domain (multidimensional parallelotope). The cardinality of the shape is the number of different index combinations in the domain.

$$h \in \text{Shape} = (S_0, S_1, S_2, ..., S_{n-1})$$
$$\text{Card}(h \in \text{Shape}) = \prod_{i=0}^{n-1} \text{Card}(S_i)$$

Tiles: We define a *Tile* as an $n-$dimensional array. Its domain is defined by a shape, and it has a number of elements of a given type, depending on the programming language chosen.

$$\text{Tile}_{\ h \in \text{Shape}} : (S_0 \times S_1 \times S_2 \times \ldots \times S_{n-1}) \rightarrow < \text{type} >$$

Our *shapes* expand KeLP regions [4] with stride domains, and unifies Chapel dense and stride domains [5] in a single type.

2.1 Library overview

The Hitmap library implements functions to efficiently create, manipulate, map, and communicate hierarchical tiling arrays for parallel computations. The library supports three sets of functionalities:

Tiling functions: Definition and manipulation of arrays and tiles, in a tile-by-tile basis. These functions can be used independently of the others, to improve locality in sequential code as well as to generate data distributions manually for parallel execution.

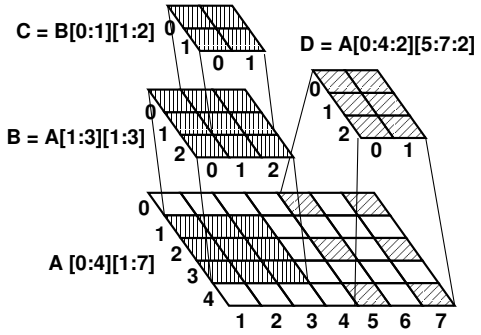

Fig. 1. Tiling creation from an original array.

Mapping functions: Data distribution and layout functions to automatically partition array domains into tiles, depending on the virtual topology selected. These functions are oriented to data- and task-distribution on parallel environments. The input needed at this point are the virtual topology and layout functions to be used, and the data structure to be distributed. These functions return (i) the ranges of the tiles that needs to be created (using the Tiling functions), (ii) the mapping between tiles and virtual processors, and (iii) the neighbor information, encapsulated in a single structure.

Communication functions: Creation of reusable communication patterns for distributed hierarchical tiles. These functions are an abstraction of a message-passing model to communicate tiles among virtual processors, and may be used with the mapping information (mapped tiles, neighborhood information, and virtual topology), to create mapping-dependent communication patterns. They return a handler that can be used to repeatedly communicate tiles among processors.

2.2 Tiling functions

Arrays and tiles are implemented with an abstract data type: *HitTile*. To use a new array, it should be declared first as a *HitTile* variable, providing its dimensions and index ranges, as in A[0:4][1:7]. This information constitute the *domain* of the array. See array A in Fig. 1, where we use a notation to specify dimensions and ranges that resembles Fortran90 and MATLAB conventions.

A new tile can be derived from another tile, specifying a *subdomain*, which is a subset of the index ranges of the parent tile. A *subtile* is indeed a tile with the same properties. At this point, the user can access the elements of the original array 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. 1, that are surrounded by local coordinates. We provide different functions to access elements and/or specify new subtiles using any of both coordinates system. Tiles may also select subdomains with *stride*, transforming regular jumps in the original array indexes to a compact representation in tile coordinates. See array D in Fig. 1.

Tiling allocation. Tiles are not automatically allocated. Instead, we provide a function that allocates memory for a tile on demand. Accesses to a tile which has been already allocated are solved referencing its own memory, no matter if the coordinates system being used is the one belonging to the tile or to its ancestor. Accesses to tiles without their own memory are mapped to the *nearest ancestor* with allocated memory. The corresponding indexes are mapped transparently to the ancestor coordinates. 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 directly or even recursively based on it. Only the subtiles which are needed locally should be allocated, while *all* the tiles generated keep their own tiles and array coordinates. Tiles with allocated memory never lose the reference to the parent tile or array. Thus, the library provides functions to update data elements of tiles with allocated memory with elements from their ancestor, or vice-versa. This is useful to create buffers or shadow copies for temporal use.

Tiling overlapping and range extension. Given an initial tile, it is possible to define two children tiles whose indexes *overlap*. Allocating overlapping tiles is a natural and easy way to generate local buffers for parallel stencil-based algorithms, where each cell should be updated taking into account its neighbors [11]. Hitmap also allows to define tiles that extend out of the range of the original array. Those elements outside of the original range can not be accessed unless memory is allocated for the whole tile. Combining overlapping and extended tiles it is easy to implement boundary conditions and stencil operations in finite-element methods.

Multilevel tiling. The mechanisms shown above allow to create hierarchies of tiles with contiguous or regular-stride subselections. The access time to elements at any level of the tile hierarchy is uniform. For more generic, irregular tile hierarchies, or other more complex data structures, it is possible to define tiles with elements that are also *HitTile*. These "supertiles" store arrays of pointers to other tiles. We have used them to store matrices by blocks, a solution useful for several linear algebra programs [12]. The access time to elements of such a tile may be non-homogeneous, as adjacent indexed elements may be allocated at different referenced tiles, belonging to different levels on their respective hierarchies.

2.3 Mapping functions

Hitmap encapsulates all partition and mapping logic into separated and reusable modules, avoiding the need to reason in terms of the number and identification of physical processors from the application code. One of the key characteristics of Hitmap is that clearly splits the mapping process in two independent parts, which have been observed to be related to different parallel algorithm features. *Topology* functions create virtual topologies using internal data, thus hiding the physical topology details. Data partition is done by a *Layout* function, that distributes domain indexes on a given virtual topology. The combination of a topology and a layout function automatically organizes the physical processors in a virtual topology, and automatically assigns an index domain part to the local virtual processor.

Hitmap provides a plug-in mechanism to select virtual topologies just by name. The plug-in functions use the information about the physical architecture and processor topology available internally. The purpose is to move the reasoning in terms of physical processors from the programmer to the library. Functions already available in Hitmap implement different topologies, such as grids of processors in several dimensions, or processor clustering depending on the processor capabilities for heterogeneous architectures. This include, for example, the use of a single, powerful processor to simulate several virtual processors. All virtual topologies in Hitmap define a group of active processors, that may be hierarchical or even recursively split.

Data partition is done by *Layout* functions that automatically compute tiling information for a given index-domain and virtual topology. An example can be found in the Sect. 7 of the supplementary material. The layout building process also creates a more sophisticated neighborhood concept, taking into account that not all virtual processors may have been assigned data to compute. This solution allows neighbor communications to skip unassigned virtual processors transparently. This happens, for example, in V-cycle iterative PDE solvers, such as MG program in the NAS Parallel Benchmarks [13], [14].

2.4 Communication functions

The Hitmap library supplies an abstraction to communicate selected pieces of hierarchical structures of tiles among virtual processors. We provide a full range of communication abstractions, including point-to-point communications, paired exchanges for neighbors, shifts along a virtual topology axis, collective communications, etc. 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.

The information needed to issue the real communication among physical processors is stored in a single data type named *Comm*. Abstract communication objects may be grouped in another data type named *Pattern*, generating reusable combinations of communication structures.

2.5 Combinations of topology and layout functions

The correctness of some parallel algorithms depends on constraints on the topology or layout functions. For example, Cannon's algorithm for matrix multiplication is designed to work with a perfect-square mesh of tasks, since the topology must have $N \times N$ virtual processors.

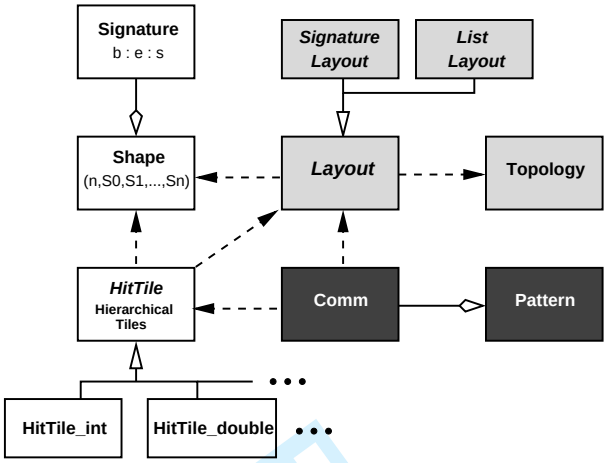

Fig. 2. UML diagram of the architecture of the Hitmap library.

But the data-pieces produced by the partition of the matrices may have any kind of block shape. For this algorithm, the chosen layout function will surely have an impact on the performance, but not on correctness. A complete implementation of Cannon's algorithm can be found in Sect. 12 of the supplementary material. On the other hand, the algorithm of the NAS MG benchmark computes a 3-dimensional stencil convolution, leading to a 3D partition and communication pattern. However, this partition and pattern may be mapped on different 1D, 2D, or 3D arrangements of processors. In general, when an algorithm is tied either to a specific topology or to a specific layout function, the other one can be freely changed. Hitmap allows to test new topology/layout combinations with little effort, by changing only the name of a plug-in function in the whole code.

Regarding robustness issues, expressing communications in terms of layouts greatly helps the programmer to build deadlock-free communication patterns. Moreover, the automatic marshalling for tile communications helps to avoid programming errors.

3 Hitmap Design and Implementation

In this section we will dive into the design and implementation details of the Hitmap library. Although Hitmap has been designed with an object-oriented approach, the current implementation has been written in C, to better exploit the original author's expertise in classical C compiler optimizations. The development of a C++ object-oriented interface is a straightforward effort.

Figure 2 shows the UML classes diagram. Recall that the UML diagrams show dependencies as dotted arrows, where the arrow points to the class used, or referenced by, the class located at the tail of the arrow. Diamond headed arrows indicate that objects of the class located at the head of the arrow are composed by several objects of the tail class. The lines with a regular white arrow head indicate classes inheriting from an abstract class, whose name is depicted with italic font. The classes in white boxes implement tiling functionalities; the classes in light gray implement layout functionalities; finally, the classes in dark gray implement communication functionalities.

3.1 Tiling classes

The classes *Signature*, *Shape*, and *HitTile*, define three data-types to support multidimensional and hierarchical tiling arrays. The library includes a macro function to specialize the *HitTile* data structure for any base type. The *HitTile* constructor only defines the tile domain. Available methods include data allocation, access to elements, and creation of tiles that represents sub-selections of the original domain. A detailed example is shown in Sect. 8 of the supplementary material.

3.2 Data partition and mapping subsystem

Light gray classes (*Topology* and *Layout*) are used for data distribution and mapping. *Topology* and *Layout* are designed as abstract classes. Each new topology function or data partition technique is implemented as an extension of one of these classes, providing the behavior of its abstract methods. A header file and a skeleton code in C are provided for both topologies and layout functions. Thus, new techniques may be easily programmed and compiled externally to the library, using them as plug-ins at compile time.

3.3 Topologies

Topologies are used from the application code invoking the constructor-like function *hit_topology (<name>)*. It receives only one parameter indicating the name of the chosen plug-in. The Topology class has only one abstract method. A new topology plug-in is implemented as a C function with a special name prefix. Topology functions receive an internal *HitPTopology* (physical topology) structure, containing information and details about the physical processors and the platform. This information is either automatically obtained by the library during initialization (e.g. querying MPI about the number of available processors and the local processor identifier), or provided statically in a platform configuration when an automatic query can not provide this information (e.g. the information about relative computing performance of each processor). The topology function fills up and returns a *HitTopology* structure. Currently, this structure supports mesh topologies for any number of dimensions. In Sect. 9 of the supplementary material we show an example of a topology plug-in code.

3.4 Layouts overview

Data-partition and mapping is done by classes inherited from the *Layout* class. Hitmap introduces a generic layout creation function, *hit_layout()*, that divides a domain, expressed by a shape, into subdomains mapped to

virtual processors. This function receives at least three parameters: (a) the name of a plug-in that implements the particular layout to be used; (b) a virtual topology created with *hit_topology()*; and (c) a shape, either created on the fly or extracted at runtime from any tile, representing its data size. Each layout function may define further compulsory parameters if needed. The *hit_layout()* function returns a *HitLayout* structure which may be queried for the generated mapping information. If there are less domain elements than processors on any dimension in the virtual topology, the layout transparently determine the active virtual processors and assigns domain elements to virtual neighbors, without manual intervention.

The Layout class defines a common interface for both regular and irregular data-partition techniques. We define two different inherited abstract subclasses for layout functions implementations. *Signature Layout*s are more appropriate for regular partitions, because they describe the relationship between processors and data indexes using signatures. *List Layout*s are more appropriate for irregular partitions. Instead of using signatures, they implement generic mapping algorithms that associate lists of indexes to processors. The latter are more generic, but not as efficient as shapes to represent big quantities of indexes organized in regular (signature) form.

3.5 Layout plug-ins implementation

To define a new signature plug-in, the programmer usually needs to provide only one function to compute the partition in one dimension. The function fills up an output signature with the local part for the local processor, and returns true/false to indicate whether a part has been assigned, or should be marked as non-active. For example, a generic block partition function in a signature layout plug-in receives four parameters: The local virtual processor index (or rank), p; the number of virtual processors on this dimension, P; the signature of the shape in this dimension to be divided, $S = (b, e, s)$; and a pointer to the resulting signature object, S'. The local part of the signature assigned to this processor, $S' = (b', e', s')$, is calculated according to:

$$b' = \frac{p * \text{Card}(S)}{\min(P, \text{Card}(S))}$$

$$e' = \left(\frac{(p+1) * \text{Card}(S)}{\min(P, \text{Card}(S))} - 1 \right) \cdot s + b$$

$$s' = s$$

If there are less index elements in the signature than the number of processors, the last processors remain inactive. It is necessary to add a condition to handle this situation.

After defining the layout function, the programmer should define a plug-in that uses it. This plug-in simply declares the function to be used and their properties, and calls a method, provided by the library, that applies the function either to each dimension of the input shape present in the topology, or to a selected dimension (chosen by the application programmer with an optional parameter when calling the plug-in). Therefore, the library internally handles all the interactions between the topology and the layout function.

The shape calculations implemented in this plug-in provide valid outputs for any combination of input parameters. Many parallel algorithms, both in literature and in real implementations, assume cardinalities that are powers of 2, or input shapes that are multiples of the number of processors, thus generating simpler codes for partition and communication. With our approach, this complexity is encapsulated in the plug-ins, making the general-case implementation of algorithms as simple as the restricted ones.

When some processors are deactivated by the layout function, neighbors are not simply the ones with adjacent indexes. The layout is capable of calculating the neighbor indexes, taking this information into account. We also implement an optional wrapping flag, to generate toroidal neighbor relationships. Thus, all the complexity of detecting neighbors in the general case is again encapsulated in the plug-in, not in the application code.

The *HitLayout* structure returned by the library contains information automatically generated only for the local part. Pointers to the signature and neighbor functions are also stored in the structure, to generate neighbor or non-local parts information on demand. Thus, the amount of local information is fixed, instead of growing with the number of processors, allowing a better scalability.

Writing a new signature plug-in is straightforward. It is enough to implement the corresponding formula into a layout function. List plug-ins, on the other hand, are more difficult to develop, because they are implemented using an algorithm instead of a signature formula. The *HitLayout* structure contains a C union with different internal data for Signature and List layouts. Hitmap provides a common API to query most of their internal information. The most relevant change is that signatures are internally substituted by lists that map each processor to a collection of associated indexes that do not need to follow a regular pattern.

In summary, the development of a plug-in consists in encapsulating in our abstract API the functions or algorithms that the programmer would otherwise hardwire into the application code, an improvement in terms of reusability with a negligible performance penalty.

3.6 Groups and hierarchical partitions

Layout functions may create *Groups*. A group associates a collection of virtual processors together, which are considered a single virtual processor. Layout functions can assign a part of the original shape not only to a processor, but to a group. The part assigned can be

a signature-based shape or an index list. This allows the processors in the same group to use further levels of data-partition (new layouts) on their assigned subdomain of indexes. This is useful for recursive data-domain decompositions, or for mapping small quantities of highly-loaded tasks with more inner levels of parallelism. As an example, Sect. 11 of the supplementary material shows the complete implementation of the QuickSort algorithm in Hitmap.

Each group has a leader processor. It is the current active processor of the group, the one doing the serial computations before further data partitions are used, and the one issuing communications to neighbor group leaders if needed. Group management is almost transparent to the programmer. Hitmap provides a conditional structure that executes a program block only if the local processor is the group leader in a given layout.

3.7 Topology and layout techniques currently implemented

The available topology plug-ins currently included in Hitmap are the following: *plain*, one-dimensional arrangement. This virtual topology simply enumerates the available processors. *mesh:* Arranges all available processors P into an n-dimensional mesh, for a given n supplied as parameter. It is based on a prime-factors decomposition. It tries to balance the number of processors on each dimension. If P is prime, it falls back to an $P \times 1 \times 1$ arrangement. *square:* Similar to *mesh* in two dimensions, but arranges as many processors as possible into a perfect-squared mesh, leaving the rest of processors available for other parallel routines.

Signature-based functions include several *blocks* functions, with different policies to allocate the group leaders (active processors), and a *cyclic* function. We do not need to introduce a explicit *block-cyclic* function, as it may be generated in two-levels using first a blocking function, and then a cyclic layout to distribute the blocks. In [15] we describe this composability layout property, and how to use it to efficiently implement an LU factorization.

Lists-based layouts include two techniques for load-balancing. The first one is based on the partition needed by the bucket sort algorithm, implemented in the IS NAS benchmark, where it is used to redistribute data buckets in terms of the buckets sizes. The second is a similar technique, also using extra information about loads associated to the domain indexes, but it may associate non-neighbor virtual processors to the same group to create a smoother load balance on non-symmetric systems. Both techniques for load balancing are also useful in recursive decomposition algorithms, such as Quicksort.

3.8 Communications implementation

The current implementation uses several features of the MPI communication library. Communications across virtual processors are encapsulated in the *Comm* objects. Any point-to-point or collective communication is represented by a single Comm object. A Comm object stores information about either a single operation, or a pair of send/receive operations. We provide different constructors for different communication operations. The constructors have a very similar interface with the following parameters, some of them optional for certain communication types: (a) Sending and/or receiving tile buffers (tile subselections); (b) sending and/or receiving virtual processes indexes; and (c) a Layout object with the information about neighborhoods generated by the data distribution over the virtual topology. The Layout constructor generates and stores in the layout object a particular MPI communicator that contains only the processors that have associated data domains after the domain distribution.

The pointers to the sending and/or receiving tile data buffers are stored in the object. The structure of the sending or receiving tiles is examined to generate MPI-derived data types that represent the tile data displacements. Any hierarchical HitTile subselection can be represented by a combination of *contiguous* and *vector* MPI-derived data types. Tiles with base elements that are also tiles need also to combine the previous types with *struct* MPI-derived data types. The result is a single, combined type that is committed and stored in the Comm object. Thus, buffering, marshalling, and unmarshalling of data is automatically managed by the MPI layer in the best possible way.

The programmer may directly provide values for the sending/receiving virtual process parameters. Nevertheless, Hitmap methodology encourages the use of the Layout methods for calculating neighbor processes indexes. In this way, a change in the real topology, in the policies selected for the virtual topology or layout building, or in data sizes, is automatically captured in the communication objects during their construction, storing different real processor indexes, or different MPI-derived data types for the data location.

Once built, the Comm object contains all the information to issue the real data transfer as many times as needed. The Comm class provides a method to activate the communication in synchronous (normal) mode, and two different methods to start and end the communication at different points of the code, allowing to implement an asynchronous mode.

Communication Patterns are implemented as a queue of Comm objects. The same activation modes are provided for Patterns. Associated methods simply traverse the internal queue calling the corresponding method on each Comm object.

Although the current backend implementation relies on the rich API of MPI, these functions are abstract enough to be ported to other backends. We are currently working in the implementation of Hitmap with other parallel programming models.

4 Experimental results

Experimental work has been conducted to show that the abstractions introduced by the library do not only simplify the complexity of codes, but they do not entail significant performance penalties.

4.1 Design of experiments

We have designed experiments with the following guidelines: (1) Choose parallel applications or kernels which are well-known and representative of important applications classes and programming paradigms. They present different challenges, and imply the use of different library resources. (2) Obtain or generate a manually programmed and optimized version of each application in C language to be used as reference, since some of the selected benchmarks were originally in Fortran. The Fortran codes has been manually ported to C, obtaining versions at least as efficient as the original codes. (3) Write a new code version based on Hitmap. (4) Execute both versions with the same inputs and conditions on selected machines, and (5) compare the codes and the execution times obtained.

The codes has been run on two different machines which represent two different types of common 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 a homogeneous Beowulf cluster of up to 36 Intel Pentium 4 nodes, interconnected by a 100Mbit Ethernet network. The MPI implementation used in both architectures is MPICH-2, compiled with a backend that exploits shared memory if available.

The codes are instrumented to measure the execution times, including both the execution of the main computation part, and the creation of data-partition, mapping, and communication information and structures. Thus, we may fairly compare the performance of the library and its applicability in real cases.

The benchmarks chosen include: Two programs from the NAS Parallel Benchmarks [13] (the MG multigrid program, and the IS integer sort program); an LU factorization and back-substitution solver based on the ScaLAPACK package [16]; and a matrix-multiplication kernel based on the generalized Cannon's algorithm. See Sect. 13 of the supplementary material for the rationale behind this benchmark choice.

4.2 Performance comparison

The execution times obtained for different versions of the benchmarks evaluated are shown in Figs. 3, 4, and 4. The times include the stage of computing tiling hierarchies, mapping, and communication information. According to the ScaLAPACK documentation, the ScaLAPACK's LU factorization implementation does not scale well if the interconnection network can not deliver several messages simultaneously, such as Ethernet. Thus, this benchmark is not suitable for the Beowulf cluster, and we did not carry on experiments for this program and platform. In the Geopar machine, when all 16 processors are used, the operative system, the MPI daemon, and the computations interfere with each other, producing additional context changes and cache misses on at least one core, delaying the overall computation. Thus, most applications exhibit a scalability limitation for 16 processors in Geopar. None of the experiments have led to incorrect results or runtime errors of any kind.

Results show that the use of Hitmap library does not imply a significant performance penalty, being less than 8.5% in the worst case (LU Scalapack). Results for the Cannon's matrix multiplication and the LU factorization programs show that the accesses to tile elements in Hitmap are almost as efficient as direct memory accesses. Moreover, the efficient management of MPI derived data types and reutilization of communication patterns produce positive effects.

In general, the cost of initializing Hitmap data-structures, layouts and communication patterns is similar to the cost of the manually programmed calculations in the reference versions. This cost is amortized by their reutilization across many iterations of the computation. It is remarkable that the IS program needs the computation of several different patterns on each repetition of the code. However, the performance delivered by the Hitmap and reference versions are almost the same.

Figure 5 shows a performance comparison between different MG benchmark implementations using state-of-the-art parallel programming models, with C and D input sets. The comparison include shared-memory models (OpenMP), PGAS models (UPC), and distributed-memory-based libraries (C+MPI, HTA, Hitmap). Results for the original NAS implementation, using Fortran+MPI, are also shown. To allow the comparison of shared-memory and distributed-memory models in terms of performance, all experiments were run in Geopar, the shared memory system described in Sect. 4.1. All implementations were compiled with GCC and equivalent optimization flags, except HTA, that requires the use of the Intel C compiler. MG implementations that support the D input size (the biggest one that fits in the machine memory) need to be compiled using the "medium" memory model in Geopar architecture. Each bar is represents the sum of the time as measured by the benchmark, and the "additional time" spent in initializations.

From the results obtained we can draw the following observations. First, UPC delivers good performance for the C input size. However, UPC's memory footprint is three to four times bigger than in other implementations. For this reason, the UPC implementation for the D input size does not fit in Geopar's memory. Second, the use of HTA to execute MG with the D input set leads to much higher execution times and "unsuccessful" results,

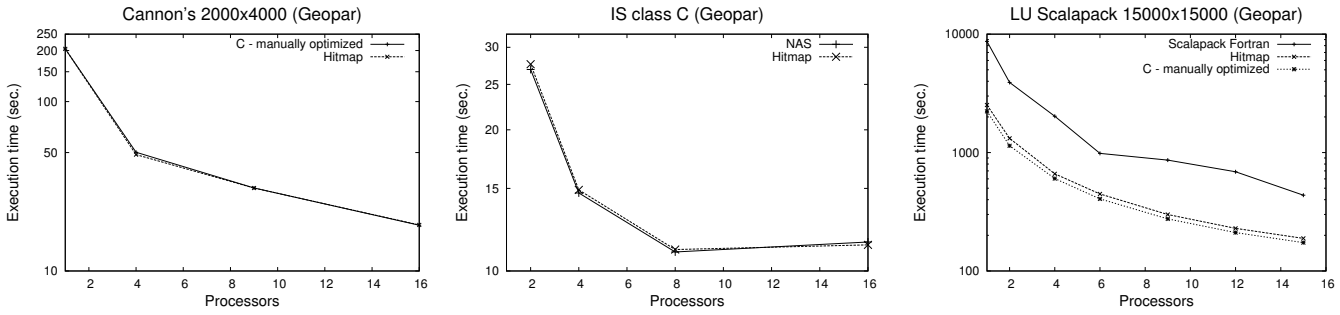

Fig. 3. Performance results for some representative parallel kernels and benchmarks in Geopar, a shared memory system. Results for MG are shown in Fig. 5.

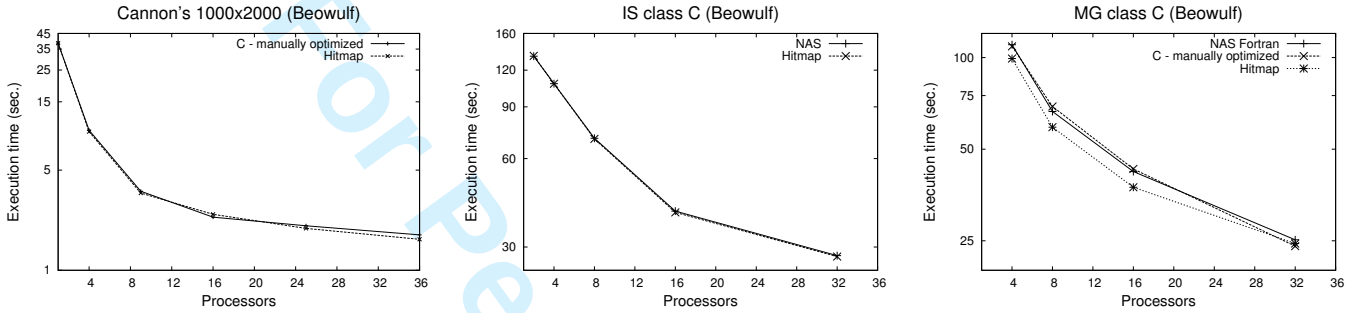

Fig. 4. Performance results for some representative parallel kernels and benchmarks in a Beowulf cluster.

so performance values for class D are not shown. Third, although C+MPI version is generally faster than the Fortran+MPI version, it does not beat the performance obtained with the OpenMP version. However, OpenMP can not be directly used in distributed memory environments.

Finally, the results show that Hitmap performance is comparable with the performance with C+MPI and generally faster than HTA, thanks to a more efficient communication management. Besides this, Hitmap presents a more flexible interface and a lower development effort, as we will show in the following section.

4.3 Development effort: code lines

To compare Hitmap code complexity with respect to manual implementations, we will use several complexity and development effort metrics, including number of lines of code, McCabe's development effort, and cyclomatic complexity.

Figure 6 shows a comparison of the Hitmap version of the benchmarks considered with the manual implementations in terms of lines of code. The comparison separates the lines devoted to parallelism (data layouts and communications), sequential computation, declarations, and other non-essential lines (input-output, etc).

With respect to MG, our results show that the use of Hitmap library leads to a significant reduction in the number of lines, specially those devoted to parallelism (partitioning and communication), even including the 14 lines consumed by the new layout plug-in developed for this example (see [14] for the details). Although MG uses a multilevel data partition, Hitmap automates the generation of communication patterns and hides the particular cases that occur in smaller grids.

Significant reductions are also obtained for Cannon's algorithm, because it only uses a single communication pattern that is derived directly from the data partition. Regarding LU, the use of Hitmap leads to a compact representation of blocks of tiles, thus reducing many computations needed to handle size, paddings, and block-cyclic distribution management. Moreover, the use of layouts hides to a great extent the details on how to build LU's complex communication patterns. Finally, IS presents smaller reduction ratios, because most of the code is sequential. Even so, lines devoted to parallelism are reduced by 38%.

4.4 Development effort: other metrics

Table 1 shows the McCabe's cyclomatic complexity (McCabe's C.C.) [17] for the benchmarks considered. This metric indicates the total number of execution paths in a piece of code. As can be seen in the table, cyclomatic complexity is greatly reduced for Cannon's (58.82%), MG (49.04%), LU (31.49%) and IS (34.52%) code. The reason is that Hitmap hides many decisions to the programmer, thus avoiding unnecessary conditional branches in the resulting code.

Table 1 also shows the the Halstead's development effort (Halstead's D.E.) [18] and the KDSI metric used in the COCOMO model [19] for the benchmarks consid-

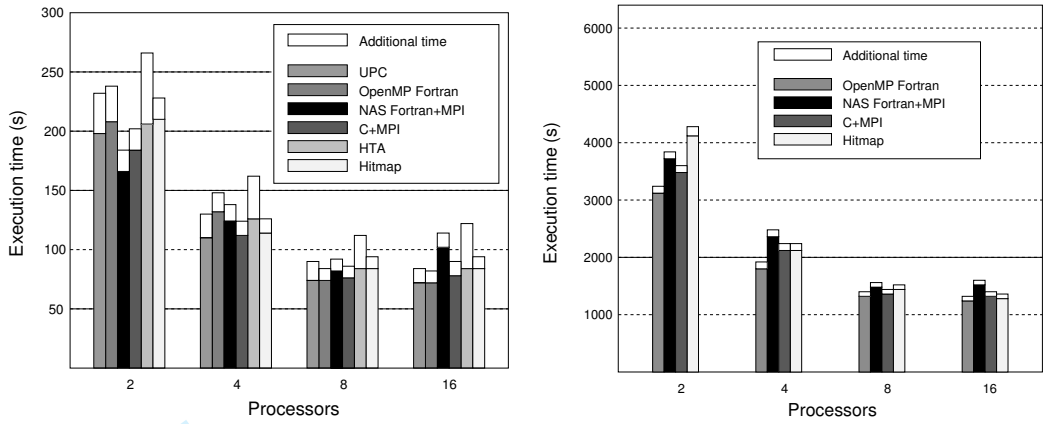

Fig. 5. NAS MG benchmark performance comparison. Class C problem size (left) and Class D problem size (right).

ered. As can be seen in the table, the use of Hitmap leads to a great reduction with all benchmarks.

We can conclude that Hitmap greatly simplifies the programmer effort for data distribution and communication, compared with the equivalent code needed to manually calculate the information used by MPI routines, or to handle the synchronization details needed in other models. Moreover, Hitmap encapsulates generic calculations into plug-ins, allowing the programmer to skip the use of tailored formula to compute local tile sizes in the application code, and neighborhood relationship at the different grain levels.

5 Related work

There is a lack of tiling support in most programming languages, with the exception of some data-parallel languages such as HPF [20] and PGAS languages such as UPC [6]. Both supply some constructors to align and distribute data among processors. HPF offers a limited set of patterns computed at compile time, while Hitmap provides a common interface that also supports data-dependent distributions computed at runtime. Moreover, HPF does not offer a truly composable distribution mechanism, since it is not possible to apply a second data distribution over the local part of a previous distribution. For example, block-cyclic distributions can not be programmed as a composition of cyclic over block distribution as in Hitmap [15]. Regarding PGAS languages, it is responsibility of the programmer to define and distribute tiles, frequently in terms of the number of processors or specific architecture details. This leads to the development of code that is hard to read and to maintain.

HTA [7], [21] is an elegant implementation of hierarchically tiling arrays in object-oriented languages as an abstract data type. Hitmap offer lower-level and more generic mapping functionalities that could be used to implement HTA as a special case. The Hitmap topology and layout plug-in system is more flexible, extensible, composable at different levels, and supports irregular or load-balancing data partitions with a common interface. These features go beyond HTA functionalities [9]. Hitmap also has a generalized hierarchy system, where a given branch of the hierarchy can be refined dynamically to an arbitrary level.

The Chapel language also proposes a transparent plug-in system for domain partitions, although no complete specification, implementation, nor experimental results are available yet [5]. Chapel proposes only one type of partition plug-ins, eliminating the flexibility to work with Topology and Layout combinations. It also forces to create specific modules for partitions which may be expressed by multilevel layouts (such as block-cyclic). Our approach allows to express communications in terms of tiles as derived from the algorithm dependencies, leading to communications of the appropriate granularity; a problem found in Chapel prototype partition modules.

Additional literature review can be found in Sect. 14 of the supplementary material.

6 Conclusions

In this paper we have presented Hitmap, an efficient library for hierarchical tiling and mapping of arrays. Hitmap presents an abstract layer, facilitating the building of more complex programming abstractions for parallel languages and compilers. Hitmap introduces a flexible tile domain and memory allocation system, featuring a modular system where programmers may add their own layout and mapping functions. It allows to build reusable and adaptative communication patterns for distributed tiles. Our experimental results show that the use of Hitmap leads to more abstract programs, easier to code and maintain, still obtaining good performance results. The Hitmap library, together with the example codes, is available at trasgo.infor.uva.es/hitmap.

Acknowledgments

This research is partly supported by the Spanish Government (TIN2007-62302, TIN2011-25639, CENIT OCEANLIDER), Junta de Castilla y León, Spain (VA094A08,

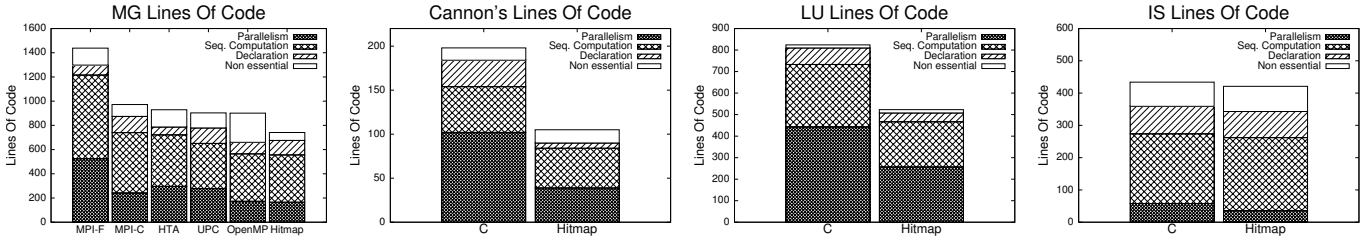

Fig. 6. Comparison of code lines.

Metric	Cannon's		MG					LU		IS	
	C + MPI	Hitmap	C + MPI	Hitmap	HTA	UPC	OpenMP	C + MPI	Hitmap	C + MPI	Hitmap
McCabe's C.C.	34	14	210	107	148	218	168	216	148	84	55
Halstead D.E.	1 892K	359K	29 568K	19 265K	54 366K	35 084K	-	27 822K	7 576K	2 683K	2 193K
KDSI (COCOMO)	201	104	1 389	945	1 277	1 413	1 343	919	606	608	496

TABLE 1
Complexity metrics and development effort for the benchmarks considered.

VA172A12-2), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative. The authors wish to thank Dr. Valentín Cardeñoso-Payo and Prof. Arjan van Gemund for their support during the early stages of this research; and Dr. Mark Bull, Dr. Murray Cole, Prof. Michael O'Boyle, Prof. Henk Sips, Dr. Maik Nijhuis, and Dr. Ana Lucia Varbanescu for many helpful discussions.

REFERENCES

[1] M. Wolfe, "More iteration space tiling," in *Proc. of the 1989 ACM/IEEE conference on Supercomputing*. Reno, Nevada, United States: ACM, 1989, pp. 655–664.
[2] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. of the ACM SIGPLAN PLDI*. Toronto, Ontario, Canada: ACM, 1991, pp. 30–44.
[3] J. Brodman, B. Fraguela, M. Garzarn, and D. Padua, "New abstractions for data parallel programming," in *First USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, Mar. 2009.
[4] S. Baden and S. Fink, "The Data Mover: A machine-independent abstraction for managing customized data motion," in *LCPC'99*, ser. LNCS, vol. 1863. Springer, 2000, pp. 333–349.
[5] B. Chamberlain, S. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.
[6] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999.
[7] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarn, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proc. of the ACM SIGPLAN PPoPP*. New York, New York, USA: ACM, 2006, pp. 48–57.
[8] A. Gonzalez-Escribano and D. R. Llanos, "Trasgo: A nested-parallel programming system," *The Journal of Supercomputing*, vol. 58, no. 2, pp. 226–234, 2011.
[9] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos, "Extending a hierarchical tiling arrays library to support sparse data partitioning," *The Journal of Supercomputing*, 2012, online-first version available.
[10] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Encapsulating synchronization and load-balance in heterogeneous programming," in *Euro-Par 2012, to appear*. Rhodes, Greece: Springer-Verlag LNCS Series, 2012.
[11] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the ACM SIGPLAN PLDI*. San Diego, California, USA: ACM, 2007, pp. 235–244.
[12] M. Castillo, E. Chan, F. Igual, R. Mayo, E. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, and F. Van Zee, "Making programming synonymous with programming for linear algebra libraries," The University of Texas at Austin, Department of Computer Sciences, Tech.Rep. TR-08-20, Apr 2008.
[13] E. Bailey, E. Barszcz, J. Barton, D. Browning, and R. Carter, "The NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep. RNR-94-007, Mar. 1994.
[14] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos, "Automatic data partitioning applied to multigrid pde solvers," in *PDP'11, to appear*. Ayia Napa, Cyprus: Euromicro, 2011.
[15] C. de Blas, A. Gonzalez-Escribano, and D. R. Llanos, "Effortless and efficient distributed data-partitioning in linear algebra," in *HPCC'10*. Melbourne, Australia: IEEE, 2010, pp. 89–97.
[16] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Society for Industrial Mathematics, 1987.
[17] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.
[18] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.
[19] B. W. Boehm, *Software Engineering Economics*, 1st ed. Prentice-Hall, 1981.
[20] D. Loveman, "High performance fortran," *Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, Feb 1993.
[21] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua, "Programming with tiles," in *Proceedings of the ACM SIGPLAN PPoPP*. Salt Lake City, UT, USA: ACM, 2008, pp. 111–122.

Arturo Gonzalez-Escribano received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2003, respectively. Dr. Llanos is Associate Professor of Computer Science at the Universidad de Valladolid, and his research interests include parallel and distributed computing, parallel programming models, and embedded computing. He is a Member of the IEEE Computer Society and Member of the ACM. More information about his current research activities can be found at http://www.infor.uva.es/~arturo.

Javier Fresno received his MS in Computer Science and his MS in Research in Information and Communication Technologies from the University of Valladolid, Spain, in 2010 and 2011, respectively. Mr. Fresno is a Ph.D. candidate at the Universidad de Valladolid. His research interests include parallel and distributed computing, and parallel programming models. More information about his current research activities can be found at http://www.infor.uva.es/~jfresno.

Yuri Torres received his MS in Computer Science and his MS in Research in Information and Communication Technologies from the University of Valladolid, Spain, in 2009 and 2010, respectively. Mr. Torres is a Ph.D. candidate at the Universidad de Valladolid. His research interests include parallel and distributed computing, and GPU computing. More information about his current research activities can be found at http://www.infor.uva.es/~yuri.torres.

Diego R. Llanos received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is a recipient of the Spanish government's national award for academic excellence. Dr. Llanos is Associate Professor of Computer Architecture at the Universidad de Valladolid, and his research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a Senior Member of the IEEE Computer Society and Member of the ACM. More information about his current research activities can be found at http://www.infor.uva.es/~diego.