Blending Extensibility and Performance in Dense and Sparse Parallel Data Management

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

Abstract—Dealing with both dense and sparse data in parallel environments usually leads to two different approaches: To rely on a monolithic, hard-to-modify parallel library, or to code all data management details by hand. In this paper we propose a third approach, that delivers good performance while the underlying library structure remains modular and extensible. Our solution integrates dense and sparse data management using a common interface, that also decouples data representation, partitioning, and layout from the algorithmic and parallel strategy decisions of the programmer. Our experimental results in different parallel environments show that this new approach combines the flexibility obtained when the programmer handles all the details with a performance comparable to the use of a state-of-the-art, sparse matrix parallel library.

Index Terms—Data partition, mapping techniques, sparse structures, parallel libraries.

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 element relationships as sparse graphs. However, not many parallel programming frameworks integrates transparently support for both dense and sparse data structures. Most parallel programing languages, such as HPF [1] or UPC [2], 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 of code developed previously for dense data structures is very poor.

In this paper we present a complete solution to handle sparse and dense data domains using the same conceptual approach. The reason is that the design of a parallel algorithm follows the same basics regardless 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 those algorithms could be simplified using high-level parallel models with abstractions for the data distribution management. Providing the appropriate abstractions to 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.

• *Departamento de Informática, Edif. Tecn. de la Información, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain. E-mail: {jfresno, arturo, diego}@infor.uva.es.*

To validate our solution, we have integrated sparse and dense support within a single communication and tiling library, using the same interface for both of them. The library currently supports dense and sparse matrices and graphs, and can be easily 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.

Our implementation is built upon the Hitmap library [3], initially designed to manage dense data structures. Hitmap performs highly-efficient data distributions and aggregated communications, expressed in an abstract form. The new version integrates dense and sparse data structures for matrices and graphs using a common interface, taking advantage of the automatic data distribution and communication functionalities provided by Hitmap. We use abstractions to properly encapsulate the management of domains, data arrays, partition techniques, and communication building, in terms of locality and neighborhood properties.

To show the application of this approach, we use two different benchmarks: A simple sparse matrix and vector product, and a real FEM method that uses a state-of-the-art graph partitioning technique. For these codes we have developed three versions: A manual C-MPI implementation; a Hitmap-based implementation; and a version that uses the PETSc parallel library. Our experimental results for both shared- and distributed-memory environments show that the use of our solution greatly reduces the associated programming effort while keeping a performance comparable to the other versions.

The paper is organized as follows: Section 2 briefly describes some related work in the field. Section 3 discusses our approach to conceptually integrate dense and sparse domains in a single parallel programming methodology. Section 4 provides an overview of the original Hitmap

library. Section 6 explains the design of our proposal to integrate dense and sparse support in Hitmap. Section 7 discusses the design and implementation of a multiplication benchmark and a real FEM application to show how the new Hitmap programming approach works. Section 8 presents experimental work related to the applications. Finally, Section 9 concludes the paper.

2 Related work

There are many tools designed to partition sparse domains, such as Metis [4], Scotch [5], or Party [6]. These tools can be used in the context of traditional programming models like MPI [7] or OpenMP [8]. However, these models require the programmer to manually code many run-time decisions based on data partition results, adapting synchronization and/or communication to the variable sizes and inter-dependencies generated.

Regarding specific parallel libraries for sparse domains, such as OSKI [9] or PETSc [10], 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 shows several differences. First, PETSc offers only one partitioning scheme where rows are distributed among processors [10]. 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, and in this paper we show how a specific graph bisection partition can be added as a new plugin. Second, PETSc is designed 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 an all-purpose library that allows to program generic applications with any kind of structured or dynamic data types, allowing the user to access the elements directly. This allows to implement other applications, such as lexicographic sorting of strings, local DNA sequence alignment, etc., without modifying the library. In [3] 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 it will be dependent on the particular data structure type needed for the application [10].

There are few proposals that use a unique representation for different kinds of domains. Chapel [11], 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.

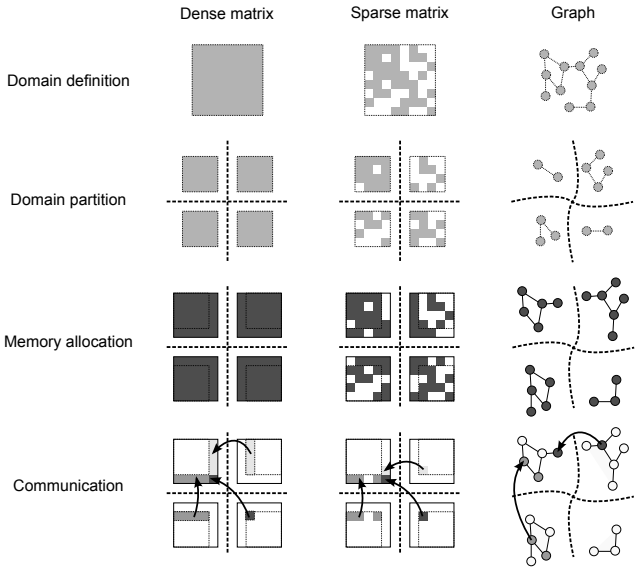


Fig. 1. Initialization stages of parallel computations for different data structures.

The original Hitmap library share some concepts with HTAs (Hierarchically Tiled Arrays) [12], 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.

3 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 programing methodology.

Regardless the dense or sparse nature of the data, most parallel programs follow the same strategy, with well-known stages. Fig. 1 shows these stages for different data domains. As can be seen, the stages are conceptually independent of the data domain, the partition technique, and the specific algorithm applied to the data.

We will use several abstract mathematical entities to describe the stages involved in most parallel programs. In the following sections, we 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 indexes. 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 the 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 virtual topology $L(D, V) : D \rightarrow V$.

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. Respectively, *Graph tiles* are defined for 2-dimensional domains. In this case, 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 They can be defined as abstract objects in terms of neighborhood relationships and 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 model 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. The previous abstractions for data management allow to define abstract explicit communications.

4 THE ORIGINAL HITMAP LIBRARY

Hitmap [3], [13] is a library for hierarchical tiling and mapping of dense arrays. It is based on a distributed SPMD 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.

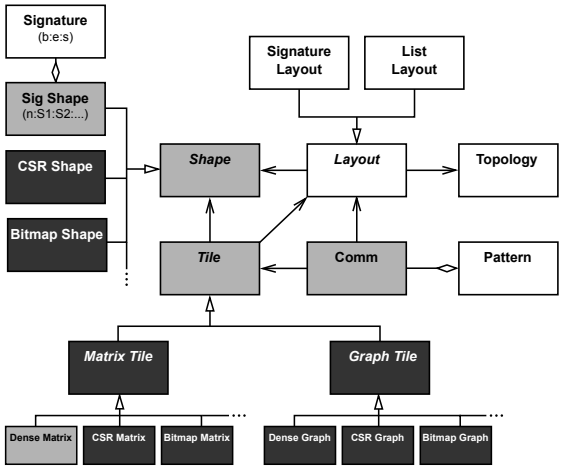


Fig. 2. Hitmap library architecture. Black boxes represent the new classes introduced to integrate dense and sparse data support. Gray boxes represents classes that had to be redesigned to deal with the new abstractions.

4.1 Hitmap architecture

Before introducing our new abstractions for sparse data structures, this section describes the architecture of the original Hitmap library. Hitmap was designed with an object-oriented approach, although it is implemented in C language. The classes are implemented as C structures with associated functions. Fig. 2 shows a diagram of the library architecture, where the white and gray boxes represent the original Hitmap classes. A summary of the basic Hitmap library API is included in the on-line, supplementary material.

In the previous Hitmap release, there was only support for dense domains, represented by a single *Shape* class. A shape represented a subspace of array indexes defined as an n-dimensional rectangular parallelotope. Its limits were determined by n *Signature* objects. Each *Signature* is a tuple of three integer numbers $S = (b, e, s)$ (begin, end, and stride), representing the indexes in one of the axis of the domain. Signatures with $s \neq 1$ define non-contiguous yet regular spaced indexes 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 indexes 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* maps actual data elements to the index subspace defined by a shape. New allocated tiles internally use a

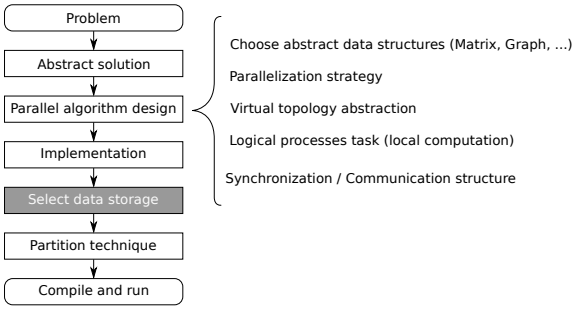


Fig. 3. Hitmap programming methodology. The gray box represent a new decision phase introduced in this paper.

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 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 *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. This class encapsulates point-to-point communications, paired exchanges for neighbors, shifts along a virtual topology axis, classical collective communications, etc. 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 composed in reusable *Patterns* to perform several related communications with a single call.

5 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. Decisions taken at any phase only affect subsequent phase.

5.1 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 interchanging information across a virtual topology of unknown size. The first step is to select the virtual topology type appropriate for the particular parallel algorithm. For example, it could be a rectangular topology where processors have two indexes (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 domains intersections.

5.2 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 communication structures designed. 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 an generic code that is adapted at run-time 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 name technique without affecting the rest of the code.

6 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 analyse the conceptual and design changes required in the Hitmap architecture to integrate sparse domains and their deeply 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.

The gray box in Fig. 3 represents the new decision phase that we have added to the Hitmap programming approach. The new classes and methods of the Hitmap API to support the sparse domains are summarized in the on-line, supplementary material.

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.

6.1 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, check if an element is inside the domain, etc. The old Shape class, based on Signatures, is transformed in a different class which implements the new Shape interface.

Sparse domains can be implemented in different ways, most of them related to traditional sparse matrices formats (COO, CSR, LIL, etc.) As long as the methods to retrieve or locate data are not in the Shape interface, some formats lead to the same Shape implementation. The differences will be found in the data-localization functions in the Tile. The old Signature Shape implementation is very efficient to locate dense information, but does not directly support a proper representation for dynamically adding or eliminating particular indexes. 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 to represent particular holes in the index domain.

We have included classes for two new kinds of sparse domains. As an example of traditional sparse domain representations, we have selected the Compressed Sparse Row (CSR) format. It is a simple and general format for sparse arrays, with minimal storage requirements [14]. 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).

As a second example of sparse domain implementation, the BitmapShape class uses a bitmap structure to represent the existing and non-existing indexes of a rectangular parallelotope. While Signature shapes only need a memory space in the order of the number of dimensions, 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, bitmap shapes are more efficient than CSR shapes in terms of memory footprint. A more complete comparison can be found in Section 3 of the supplementary material.

Comparing bitmap shapes with Signature shapes, bitmap structure is almost as efficient as the Signature shape to retrieve data by coordinates. There is a small performance penalty due to the extra arithmetic operations involved in the bitmap check before accessing the data.

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. Thus, building an expanded shape implies to traverse the local shape once. The result is another shape that can be used along with the layout object to compute intersections with neighbor subdomains, automatically determining an efficient marshalling scheme for communications. These functionalities allow to program neighbor synchronization codes with graph partitioning, in the same way than for dense matrices.

6.2 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. In general, the implementations should keep the data in a single block of contiguous memory, using index transformations and other ancillary structures to locate them in memory. This also help to implement efficient functionalities to traverse and communicate tiles. It is possible to create different Tile implementations for the same kind of Shape.

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 allows 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. Edges information and vertices values can be stored in different, internal data structures (matrix and vectors, respectively), with different access methods. It is easy to create different Tile implementations for matrices, graphs, or other data structures, based on the same Shape implementation (see Fig. 2).

We have developed new Tile implementations for matrices and graphs, with both dense and sparse domains. As it is shown in Fig. 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. The mapping functions to be implemented on the tiles are directly derived from the CSR format definition. In CSR, as in some other sparse data representations, retrieving data elements using their indexes 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 these 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 description of the Hitmap sparse iterator can be found in the supplementary material.

6.3 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 reject non-appropriate kinds of shape, or be 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. 3).

As an example of partition/mapping techniques appropriate for sparse domains, we have integrated one of the techniques found in Metis [4] into a new plug-in. Metis is a state-of-the-art graph partitioning library based on bisection methods. Our plug-in translates the information retrieved from the Shape object to the exact array format expected by the Metis function. After this translation step, the plug-in calls Metis, providing it also 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.

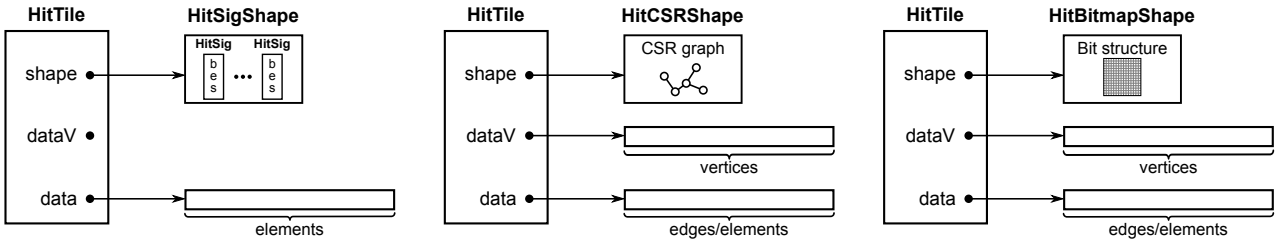
6.4 Communication

There is no change in the methods definition of the Communication class. However, the original Communication class contained a private method that used the Signature Shape of the tile to generate an MPI derived data type. This type 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 has been moved to the Tile interface and Tile implementations, and they are called from the Communication class when needed.

The new version of the Communication 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. Other utilities have been added to the library for reading and writing sparse data tiles in different formats, like the Harwell-Boeing format [15], or plain CSR format.

7 Benchmarks

In this section we describe two benchmarks developed to evaluate the dense and sparse data integration in the Hitmap library. The first one is a simple sparse matrix and vector multiplication, a benchmark that allow us to compare Hitmap with efficient implementations using well-known tools like PETSc. The second 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 depends on the (sparse/dense) structure of the input data. We have also implemented this second benchmark in PETSc, where many of these features that Hitmap offers has to be done manually.



(a) Matrix tile using a Signature Shape. (b) Graph tile using a CSR Shape. (c) Graph tile using a Bitmap Shape.

Fig. 4. Internal structure of example tile objects with different Shape implementations.

7.1 Sparse matrix-vector multiplication benchmark

The first benchmark is a simple matrix-vector multiplication $y = Ax$, where the A matrix is sparse and the x and y vectors are dense. 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. To prevent overflow in the output vector, the matrix elements will be randomly initialized to satisfy $\sum_{j=1}^{n} A[i][j] \leq 1 \, \forall i$.

7.2 FEM benchmark

We have chosen a benchmark that calculates the equilibrium position of a 3-dimensional spring system, represented as a graph. 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. The mathematical background of this benchmark is explained in the on-line, supplementary material.

8 Experimental results

Experimental work has been conducted to show that the abstractions introduced by the library simplify the complexity of codes that deal with the sparse domains manually, without introducing significant performance penalties.

We have developed three parallel programs for each benchmark. The first one codifies an *ad-hoc* implementation of the CSR format to represent a sparse matrix or a graph, manually dealing with the calls to Metis partitioning and MPI communications. The second one is an equivalent program written with the Hitmap library. The third implementation is made with the PETSc library, using sparse PETSc matrices representation, and the solvers for linear systems included in the library.

The Hitmap kernel code is not dependent upon the Tile subclass used. Therefore, different extensions of the Tile class can be used just by choosing a different name, without affecting the rest of the code. This feature allowed us to use the same code to experiment with any of the three Shape implementations, in order to compare their efficiency for different graph densities.

We use three different experimental platforms with different architectures: A multicore shared-memory machine, and two distributed clusters of commodity PCs. 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. 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, 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 first cluster each process is executed by a different core. But, in the second cluster, the mono-core processors execute more than one process when there are more processes than nodes.

8.1 Performance

To test whether the new Hitmap abstractions introduce performance inefficiencies, we measure and compare the performance of the two benchmarks described with the following implementations: (1) the MPI code manually developed and optimized; (2) the Hitmap version that uses the CSR Shape implementation; and (3) the PETSC version. CSR was chosen because it is the most efficient representation for this problem. (An experimental comparison between CSR and Bitmap implementations can be found in the on-line, supplementary material.) We have conducted experiments with matrices and graphs. 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 [16] representing typical graphs modeling real 3-dimensional structrures, with very low density degree. We also experiment with random graphs generated using the PreZER

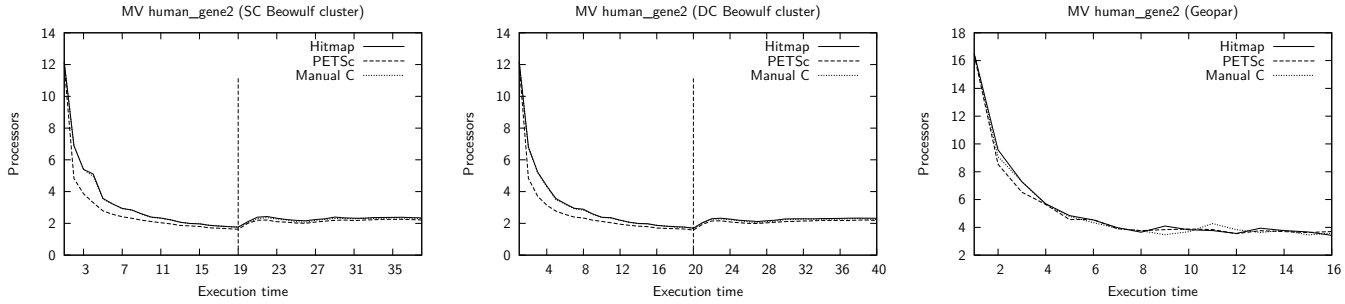


Fig. 5. Execution time comparison for Hitmap, manually developed, and PETSc implementations of the MV (matrix-vector multiplication) benchmark using a representative inputset.

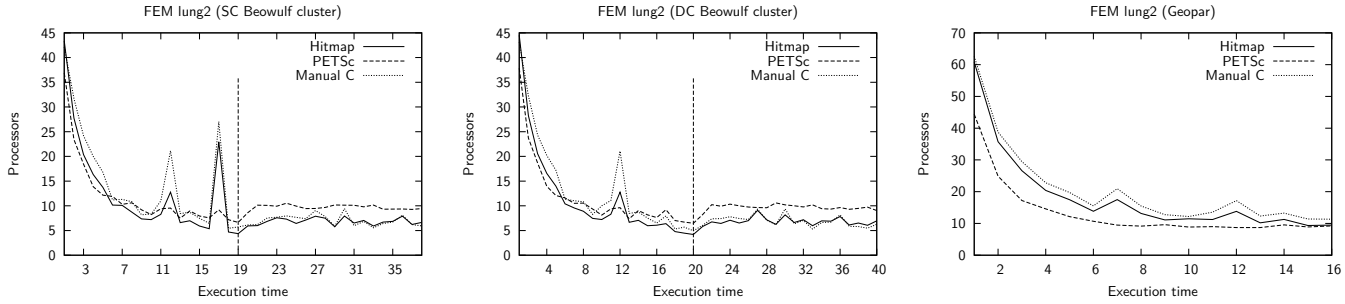


Fig. 6. Execution time comparison for Hitmap, manually developed, and PETSc implementations of the FEM (Finite Element Method) benchmark using a representative inputset.

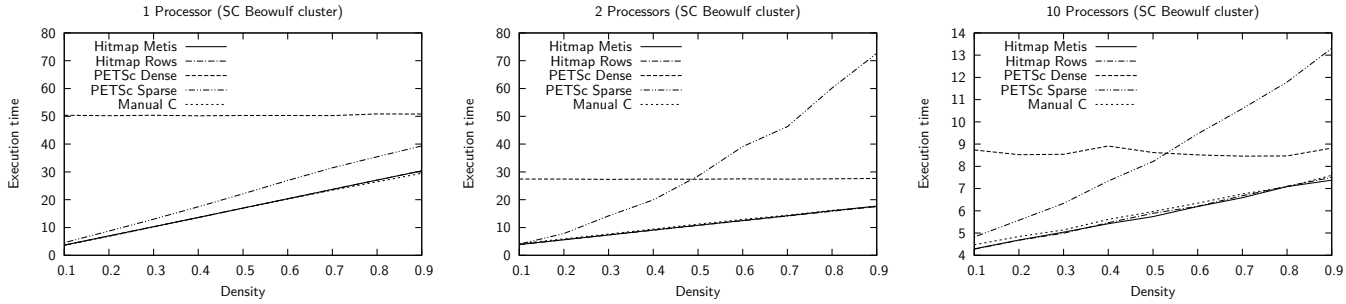


Fig. 7. Execution time comparison for Hitmap (Metis and Row partition), manually developed, and PETSc (Dense and Sparse data structures) implementations of the FEM (Finite Element Method) benchmark using random generated matrices with different densities in the in the single-core cluster.

implementation of the Erdõs-Rnyi $\Gamma_{v,p}$ model [17]. 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 will now present results for a representative input set of each type. Section 5 of the supplementary material includes a more comprehensive study using more examples with different graph characteristics.

The matrix-vector multiplication benchmarks have been run using 100 iterations of the algorithm explained in Section 7.1. For the spring benchmark we have fixed the number of iterations to end the computation, 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.

Fig. 5 shows the execution times (without initialization times) obtained for the matrix-vector multiplication benchmark in the three architectures for a representative inputset. For this benchmark, the times for all versions with one process are the same. PETSc obtains slightly better performance for a small number of processors, but the scalability trend is the same for all versions when the number of processes grows, specially when it approaches the number of nodes in the cluster and the

communication costs are highly noticeable.

Figure 6 shows the execution times of the FEM benchmark, in the three architectures considered for a representative graph example. The manual C and the Hitmap versions present some irregularities for some 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 than 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 oversubscribing the system with more processes than nodes, the performance of PETSc degrades while Hitmap maintains the behavior of the manual version.

Finally, Fig. 7 shows the execution times obtained with the FEM benchmark for different input graph densities. For this experiment set we compare two additional versions: Hitmap using a row partition instead of Metis and PETSc using its dense matrix format. The figure shows the results in the single-core cluster for 1, 2 and 10 processors. 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 manual version and they have better performance than 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.

8.2 Code complexity

To compare Hitmap code complexity with respect to the parallel manual and PETSc implementations, we present several code complexity and development-effort metrics.

For all three versions, we analyze the code containing the upper abstraction level and the solvers that implement the parallel algorithms. In the case of PETSc they are included into the library as functions called directly at the top level. Internal functions of Hitmap and PETSc devoted to implement data partitions, data accesses, marshaling, communications, etc, are skipped in this analysis. We summarize here the main results. The full details of this comparison can be found in Section 4 of the supplementary material.

For the matrix-vector multiplication benchmark, Hitmap reduces the total number of code lines in 56% comparing with the manual implementation and 37% comparing with PETSc. Moreover, the McCabe's cyclomatic complexity shows reductions of 66% and 39% respectively. It is remarcable that lines specifically devoted to parallelism control are reduced in 87% with respect to the manual version.

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. The cyclomatic complexity for this benchmark shows reductions of 54% and 73%. Lines specifically devoted to parallelism control are reduced in 88% compared with the manual version.

Using Hitmap abstractions greatly simplifies writing and maintaining a parallel program comparing with manually hardwiring 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, a task that implies a lower development effort in Hitmap.

Some extra code complexity in PETSc comes from the limited support of data types. The array structures in PETSc are always composed of single floating point elements. 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. The Hitmap implementation of the FEM benchmark uses 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 of the solver codes and the internal data representation in the tile subclass, and easily changed with the plug-in system. The flexibility and versatility of the Hitmap approach allows to introduce new data-structures, partitions, or program communication structures with minimum impact on the rest of the code.

9 Conclusions

In this paper we present an approach to integrate dense and sparse data management in parallel programming. This approach allows 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 we introduced the proper abstractions to support this approach in Hitmap, a library

for efficient partition and communication of dense hierarchical tiles. To illustrate how the approach and the tool work, we have developed a simple matrix-vector multiplication and a real FEM application using the new Hitmap library. We have used these programs as benchmarks to compare performance and programming effort metrics with respect to a manually-developed MPI code, and with 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.

We are currently working on integrating new partition techniques and data-structure formats for other applications with different synchronization structures. The Hitmap library is available under request.

ACKNOWLEDGMENTS

This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2); Ministerio de Industria, Spain (CENIT OCEANLIDER); MICINN (Spain) and the European Union FEDER (Mogecopp project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E); and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

REFERENCES

[1] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of High Performance Fortran," in *HOPL III*. New York, NY, USA: ACM Press, 2007, pp. 7.1–7.22.

[2] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," IDA Center for Computing Sciences, Tech. Rep., 1999.

[3] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos, "An extensible system for multilevel automatic data partition and mapping," *IEEE Transactions on Parallel and Distributed Systems*, vol. to appear, 2013.

[4] G. Karypis and V. Kumar, "MeTiS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse MatricesVersion 4.0," University of Minnesota, Tech. Rep., 1998.

[5] F. Pellegrini, "PT-Scotch and libScotch 5.1 Users Guide," Tech. Rep., 2010.

[6] B. Monien and S. Schamberger, "Graph Partitioning with the Party Library: Helpful-Sets in Practice," in *16th Symposium on Computer Architecture and High Performance Computing*, 2004, pp. 198–205.

[7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.

[8] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, ser. Scientific and Engineering Computation Series. Cambridge, MA: MIT Press, 2008.

[9] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005*, ser. Journal of Physics: Conference Series. San Francisco, CA, USA: Institute of Physics Publishing, June 2005.

[10] S. Balay, J. Brown, and Kris Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Users Manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.4, 2013.

[11] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in chapel: philosophy and framework," in *2nd USENIX conference on Hot topics in parallelism*, ser. HotPar'10. June 14-15, Berkeley, CA, USA: USENIX Association, 2010, p. 12.

[12] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *PPoPP '06*. New York, NY, USA: ACM Press, 2006, pp. 48–57.

[13] 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.

[14] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. SIAM, July 1994, vol. 64, no. 211.

[15] I. S. Duff, R. G. Grimes, and J. G. Lewis, "User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release 1)," Rutherford Appleton Laboratory, Didcot, Oxon, England, Tech. Rep., 1992.

[16] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *To appear in ACM Transactions on Mathematical Software*.

[17] S. Nobari, X. Lu, P. Karras, and S. Bressan, "Fast random graph generation," in *Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT '11*. New York, New York, USA: ACM Press, 2011, pp. 331–342.

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.

Arturo Gonzalez-Escribano received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2003, respectively. Dr. Gonzalez-Escribano 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.

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.