

Extending a hierarchical tiling arrays library to support sparse data partitioning

Javier Fresno · Arturo Gonzalez-Escribano ·
Diego R. Llanos

Published online: 13 April 2012
© Springer Science+Business Media, LLC 2012

Abstract Layout methods for dense and sparse data are often seen as two separate problems with their own particular techniques. However, they are based on the same basic concepts. This paper studies how to integrate automatic data-layout and partition techniques for both dense and sparse data structures. In particular, we show how to include support for sparse matrices or graphs in Hitmap, a library for hierarchical tiling and automatic mapping of arrays. The paper shows that it is possible to offer a unique interface to work with both dense and sparse data structures. Thus, the programmer can use a single and homogeneous programming style, reducing the development effort and simplifying the use of sparse data structures in parallel computations. Our experimental evaluation shows that this integration of techniques can be effectively done without compromising performance.

Keywords Data partition · Layouts · Distributed computing · Sparse data

1 Introduction

In parallel applications, the data distribution and layout is a key issue that can determine the performance and scalability. Data distribution problem is usually handled differently depending on the use of dense or sparse data structures. In the former case, there are many languages with primitives and tools to deal with data locality and/or distribution, such as HPF [1], OpenMP [2], or UPC [3]. However, sparse structure

J. Fresno · A. Gonzalez-Escribano (✉) · D.R. Llanos
Dpto. Informática, Universidad de Valladolid, Valladolid, Spain
e-mail: arturo@infor.uva.es

J. Fresno
e-mail: jfresno@infor.uva.es

D.R. Llanos
e-mail: diego@infor.uva.es

support is not usually integrated in the programming languages despite the fact that a wide range of important problems are based in unstructured graph structures instead of dense arrays. The common approach to manage sparse data is using a library. There are many libraries for partitioning graphs, meshes, and other sparse structures, such as Metis [4], Scotch [5], or Jostle [6].

There are some proposals that use a unique representation for different domains. Chapel, a new parallel language, uses a representation of indexes-set called a domain [7]. The Chapel's proposal aims at supporting domains for dense, sparse, strided, associative, and unstructured data aggregates. However, there is no full or efficient implementation, yet.

Another previous proposal [8] has evolved into the Trasgo system and the Hitmap library. Hitmap is a basic tool in the back-end of the Trasgo compilation system [9]. Trasgo offers a global-view approach with flexible and explicit mechanisms to deal with locality. The code generated by Trasgo uses the Hitmap library to perform a highly efficient data distribution and aggregated communications. The Hitmap library is publicly available [10].

In this paper, we present a new approach to hide the internal details of dense and sparse data structure, using a common Hitmap interface to deal with both types of data. Combining the dense and sparse manipulation under a common interface has great advantages. It simplifies programming by hiding the partition details in reusable and flexible plug-ins. The programmer can focus on the algorithm parallel implementation without thinking in terms of the underlying data structure. The coding of a parallel application follows the same basic pattern in both cases, using the same API for the same functionalities. Moreover, the implementation of Hitmap abstractions has been carried out taking into account further native compiler optimizations. Our experimental results show that using Hitmap greatly simplifies programming with a negligible impact on performance. This new version of Hitmap with the examples discussed in this paper is already available in the Hitmap repository.

This paper is organized as follows: Sect. 2 provides a brief overview of the Hitmap library. Section 4 introduces the benchmark that illustrates our proposal and describes its different implementations. Section 5 contains the experimental work. Section 6 discusses related approaches. Finally, the paper ends with the conclusions in Sect. 7.

2 Hitmap library

Hitmap is a highly-efficient library for hierarchical tiling and mapping of arrays [11, 12]. It aims at simplifying parallel programming, providing functionalities to create, manipulate, distribute, and communicate tiles and hierarchies of tiles. In this section, we will present the basic ideas of the Hitmap library needed for the further discussion.

Hitmap library supports functionalities to: (i) generate a virtual topology structure; (ii) map the data to a different processor with chosen load-balancing techniques; (iii) automatically determine inactive processors at any stage of the computation; (iv) identify the neighbor processors to use in communications; and (v) build communication patterns to be reused across algorithm iterations.

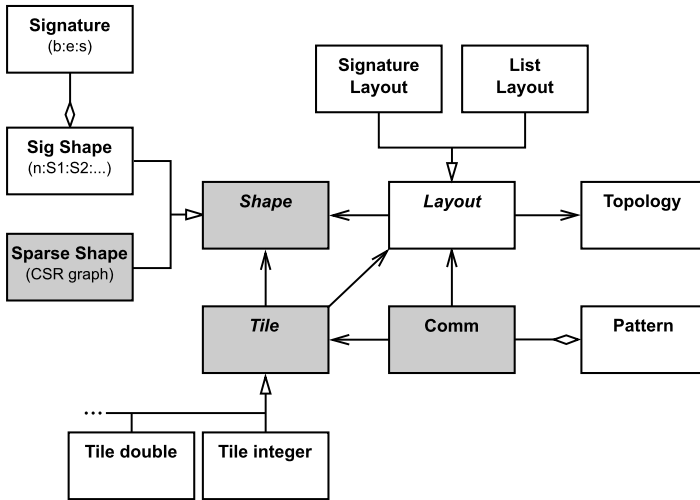


Fig. 1 Hitmap new architecture. *Shaded areas* represent the new sparse data support

Hitmap is designed with an object-oriented approach, although it is implemented in C language. Figure 1 shows a class diagram of the library architecture. The classes are implemented as C structures with associated functions. A *Signature*, represented by a HitSig object, is a selection of array indexes in a one-dimensional domain. Hitmap uses a *Shape* object to represent a domain of data. The new Hitmap version uses inheritance to integrate shape objects that represent sparse data domains. A *Tile* is an array whose domain is defined by a shape. Hitmap has functionalities to dynamically declare selections of tiles to construct tile hierarchies. A tile may be defined without allocated memory, allowing to declare and partition arrays before assigning memory to them. In this way, only the parts effectively mapped to a given processing unit are allocated.

Hitmap provides programming tools to apply different data-partition and layout techniques over automatically generated virtual topologies, hiding the details of the physical processors, topology, and mapping. Both the virtual topology generation and the partition techniques are integrated in the library as plug-in modules extending abstract classes. Programmers may include their own new techniques.

The virtual topology techniques are invoked by name with no extra parameters. They use the internal information of the target system. The result is a HitTopology object which can be queried and it is used as a parameter for data partition and layout. The layout plug-in modules allow computing a partition of a shape domain over a virtual topology. The result of a layout plug-in is a HitLayout object that contains information about the local part of the domain mapped to the current processor, the neighbor relations, and methods to return or compute on-the-fly all the information needed to exchange data. The plug-ins encapsulate the computations needed to deal with physical topology and data location at all the mapping stages; details which are usually hardwired in the code by the programmer. The combination of these plug-in systems allows the programmer to easily create ab-

struct codes which also simplifies debugging operations with any kind of data structures.

Finally, an information exchange operation can be specified creating a HitCom object. The constructor receives a HitShape to specify the data to be moved and a HitLayout object with the mapping and topology information. The result is a HitCom object with all the information needed to execute the data exchange as many times as required. Moreover, several HitCom objects can be composed in reusable communications patterns represented by HitPattern objects. The library is built on top of the MPI communication library, for portable communication and synchronization on different architectures. Hitmap internally exploits several MPI techniques that increase performance.

3 Supporting sparse data in Hitmap

The previous version of the Hitmap library was oriented to manipulate and communicate tiles of data with contiguous domains, or non-contiguous but regular index selections. In this new version of the library, we have extended the Hitmap functionalities to support sparse data structures. This allows the programmer to work with graph or sparse matrices using a similar API, and a homogeneous programming methodology. To support the manipulation, mapping, and communication of these new data structures, we needed to make several structural changes in the library.

The first change is related to the data domains represented by the HitShape class. In the previous version of the library, a HitShape object was composed of several HitSig objects, to represent a multidimensional selection of indexes. We have transformed the HitShape class into an interface, with two different implementations, one for the old dense domain and the other for the new sparse domain (see Fig. 1).

The HitSparseShape class encapsulate a sparse matrix format to represent the sparse data domain. The first sparse matrix format that we have implemented is Compressed Sparse Row (CSR). It is a well-known and widely used format for sparse data. CSR is simple, it does not make any assumptions about the matrix structure, and it has minimal storage requirements [13]. There are other formats that can offer a better performance in some particular applications. It is possible to support any of these formats with new implementations of the proposed interface.

To illustrate how to develop new layout plug-ins that make specific partitions and mapping techniques for sparse domains, we have implemented an example HitLayout plug-in. We have integrated one of the graph partition techniques of the Metis library [4]. The plug-in receives a HitShape with the sparse domain, and it calls the Metis library to compute the local part. Metis also uses the CSR format. Thus, the plug-in needs to apply a minimal data-format transformation. With the result returned by Metis, the plug-in creates a new HitShape with the local graph part. The resulting HitLayout object can optionally contain lists with vertices belonging to other processors. This information can be used to find the owner of a given vertex and to exchange data between processors.

To allow the implementation of Finite Elements Method's (FEM) applications, we have included a code to automatically compute the list of vertices by other processors which are adjacent to the local vertices.

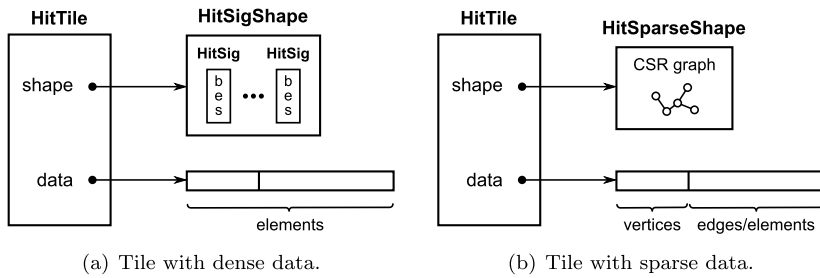


Fig. 2 Common tile object for dense and sparse data

A HitTile is the same object for dense and sparse data, as it is shown in Fig. 2. The tiles constructor receives a HitShape object. The tile constructor internally checks the type of shape. For signature shapes, we allocate memory for the elements of the array. For sparse shapes, we could allocate memory for the vertices and weight information for the edges of a graph, or we could store the values for the nonzero elements. We have also added macros and functions to easily access data and iterate across the vertices values.

In some applications, a processor needs the values of neighbor vertices mapped to other processors. The new version of the HitCom object supports a new global communication type. It is designed to exchange the data of adjacent vertices assigned to different processors. The HitCom object uses the internal information calculated in the layout to determine which vertices should be sent to any other processor, and which vertices should the current processor receive from any other processor.

Finally, other functionalities have been added to the library to facilitate the sparse data management. For example, functions for input of sparse data, like reading the Harwell–Boeing format or plain CSR format.

4 Experimental methodology

In this section, we discuss the methodology followed to create a benchmark to test the efficiency of our implementation. We select a simple problem that involves a computation over a sparse data structure.

We extend the idea of neighbor synchronization in FDM applications on dense matrices to civil engineering structural graphs; see, e.g., [14]. The application performs several iterations of a graph update operation. It traverses the graph nodes, updating each node value with the result of a function on the neighbor nodes values. To simulate the load of a real scientific application, we write a dummy loop, which issues 10 times a mathematical library operation (sin). We use as benchmarks different graphs from the Pothen group of the *University of Florida Sparse Matrix Collection* [15].

The codes have been run on Geopar, an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6 GHz and 32 GB of RAM. We have used MPICH2, compiled with a backend that exploits shared memory for communications if available in the target system.

C implementation We have first developed a serial C implementation, to have a reference for speedups and to verify the results of the parallel versions.

Our first parallel version includes the code to compute the data distribution manually. This highly-tuned and efficient implementation is based on Metis library to calculate the partition of the graph, and MPI to communicate the data between processors. The Hitmap library implementation is also based on the same tools. Thus, the comparison between performance results of a Hitmap parallel version and the manual one will show any potential inefficiency introduced in the design or implementation of Hitmap.

The manual parallel version has the following stages: (i) A sparse graph file is read; (ii) The data partition is calculated using the Metis library; (iii) Each processor initializes the values of the local vertices using a random function; (iv) A loop performs 10,000 iterations of the main computation, including updating the values of each local vertex, and the communication of the values for neighbor vertices to other processors; (v) The final result is checked with the help of a hash function.

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

In Fig. 3, we show the main function of the Hitmap code. Line 2 uses a function to read a graph stored in the file system, and returns a shape object. Then, a virtual topology of processors that uses the internal information available about the real topology is created transparently to the programmer with a single call.

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

In line 11, we obtain the shape of the local part of the graph with only the local vertices. In the following line, we use the layout to obtain an extended shape with local vertices plus the neighbor vertices from other processors. This is the equivalent to the shape of a tile with a shadow region in a FDM solver for dense matrices. This shape is used to declare and allocate the local tile with double elements (line 16).

In line 22, a HitCom object is created to contain the information needed to issue the communications that will update the neighbor vertices values. Data marshaling and unmarshaling is automatized by the communications objects when the communication is invoked (see lines 23 and 31).

Lines 25 to 32 contain the main iteration loop to update local nodes values and reissue communications with a single Hitmap call that reuses the HitCom object previously defined.

```

1 // Read the global graph.
2 HitShape shape_global = hit_shapeHBRead("graph_file.rb");
3
4 // Create the topology object.
5 HitTopology topo = hit_topology(plug_topPlain);
6
7 // Distribute the graph among the processors.
8 HitLayout lay = hit_layout(plug_layMetis, topo, &shape_global);
9
10 // Get the shapes for the local and local extended graphs.
11 HitShape local_shape = hit_layShape(lay);
12 HitShape ext_shape = hit_layExtendedShape(lay);
13
14 // Allocate memory for the graph.
15 HitTile_double graph;
16 hit_tileDomainShapeAlloc(&graph, double, ext_shape);
17
18 // Init the local graph.
19 init_graph(graph, shape_global);
20
21 // Create the communicator and send neighbor vertices.
22 HitCom com = hit_comSparseUpdate(lay, &graph, HIT_DOUBLE);
23 hit_comDo(&com);
24
25 int i;
26 // Update loop.
27 for(i=0; i<ITERATIONS; i++){
28     // Update the graph.
29     synchronization_iteration(local_shape, ext_shape, &graph);
30     // Communication.
31     hit_comDo(&com);
32 }

```

Fig. 3 Kernel code of the Hitmap version

The function that updates the local values of the graph (not shown in the code) uses two iterator of the library to traverse the edges and get the contribution of each neighbor.

5 Experimental results

In this section, we compare the performance obtained with the benchmark described in the previous section. We have tested the benchmark with different graphs from the Pothen group of the *University of Florida Sparse Matrix Collection* [15]. In this section, we discuss the results from two representative cases in the group collection: The bodyy6 graph with 19366 vertices and pwt graph with 36519 vertices. Figure 4 shows the speedup for both manual C and Hitmap benchmark implementations. There is no significant difference between the two implementations in terms of performance. Therefore, the abstractions introduced by Hitmap (such as the common interface for dense and sparse data structures, or the adaptation of the partition technique in the plug-in module system) do not lead to performance reduction compared with the manual version.

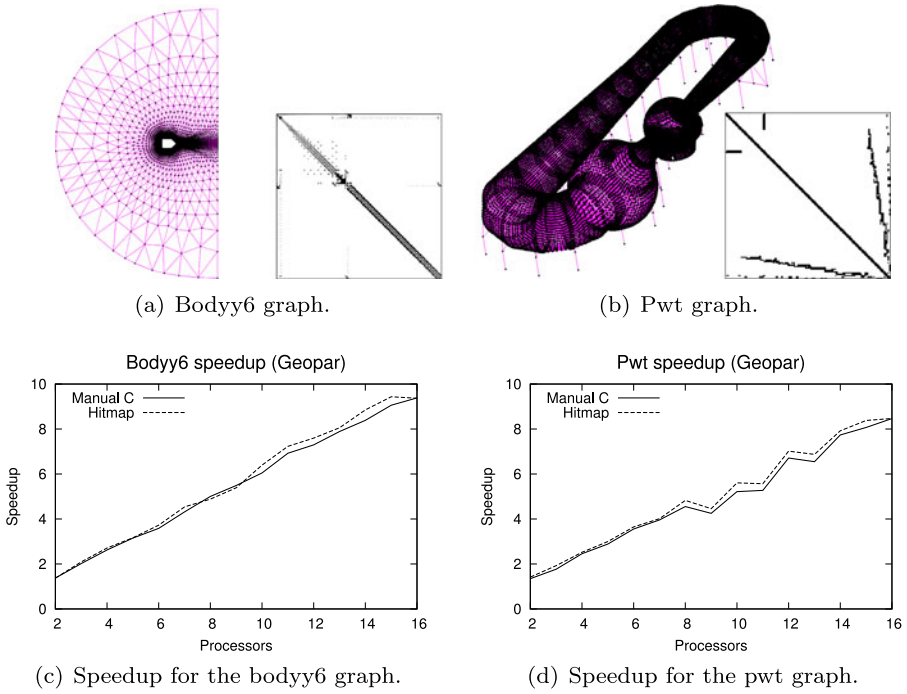


Fig. 4 Input sets and their performance in Geopar

Fig. 5 Comparison of the lines of code

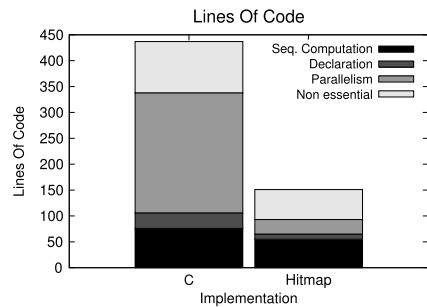


Figure 5 shows a comparison of the Hitmap version with the C version in terms of lines of code. We distinguish lines devoted to sequential computation, declarations, parallelism (data layouts and communications), and other non-essential lines (input-output, etc.). Taking into account only essential lines, our results show that the use of Hitmap library leads to a 72 % reduction on the total number of code lines with respect to the C version. Regarding lines devoted specifically to parallelism, the percentage of reduction is 78.7 %. We have also use the cyclomatic complexity metric to compare the codes. The total cyclomatic complexity of the manual version is 74 whereas the Hitmap versions has a total value of 17. The reason for the reduction of complexity in the Hitmap version is that Hitmap greatly simplifies the programmer

effort for data distribution and communication, compared with the equivalent code to manually calculate the information needed in the MPI routines.

6 Related work

There are other proposals that offer tiling support, for example, the PGAS languages, although they still leave the responsibility of defining and distributing tiles to the programmer, making the development difficult. Hitmap uses similar techniques like those of HTA [16], a library for hierarchically tiling arrays. However, Hitmap offers lower-level and more generic mapping functionalities and its topology and layout plug-in system is more flexible. Chapel [7] also proposes a transparent plug-in system for domain partitions. Since it has only one type of partition, it does not have the flexibility of Topology and Layout combinations offered by Hitmap. Chapel has also proposed a framework to integrate dense and sparse partitions. However, the solution to support sparse distributions is not developed, yet [17].

7 Conclusions

This paper shows how the support for dense and sparse data structures can be integrated in an automatic data partitioning parallel library. We have added sparse structure support in Hitmap, a highly-efficient modular library for hierarchical tiling and mapping of arrays. We have illustrated how to use the library to implement a simple graph algorithm. We have also measured the efficiency of the library in terms of performance comparing with a manual implementation. The results show that the abstraction introduced by the library does not reduce performance. We have also measured the code complexity in terms of lines of code and cyclomatic complexity. Our results show that it is possible to use a common interface for both dense and sparse data structures with a homogeneous coding style, thus reducing the associated development cost compared with manually coding the data structure management, its partition, and the communication of locally mapped subdomains when needed. As it is shown by the experimental results, this can be done without sacrificing performance.

Acknowledgements This research is partly supported by the Ministerio de Industria, Spain (CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER), Ministerio de Ciencia y Tecnología, Spain (CAPAP-H3 network, TIN2010-12011-E, TIN2011-25639), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission—Capacities Area—Research Infrastructures Initiative.

References

1. Kennedy K, Koelbel C, Zima H (2007) The rise and fall of high performance Fortran. In: Proceedings of the third ACM SIGPLAN conference on history of programming languages (HOPL III). ACM Press, New York, pp 7.1–7.22 (New York, NY, USA)
2. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in OpenMP, 1st edn. Kaufmann Los Altos

3. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K (1999) Introduction to UPC and language specification
4. Karypis G, Kumar V (1998) MeTiS—a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices—version 4.0
5. Pellegrini F (2010) PT-Scotch and libScotch 5.1 user's guide
6. Walshaw C (2002) The serial JOSTLE library user guide: version 3.0
7. Chamberlain BL, Deitz SJ, Iten D, Choi S-E (2010) User-defined distributions and layouts in chapel: philosophy and framework. In: Proceedings of the 2nd USENIX conference on Hot topics in parallelism (HotPar' 10), p 12. USENIX Association
8. González-Escribano A, van Gemund AJ, Cardeñoso Payo V, Portales-Fernández R, Caminero-Granja JA (2005) A preliminary nested-parallel framework to efficiently implement scientific applications. In: High performance computing for computational science (VECPAR 2004), pp 541–555
9. González-Escribano A, Llanos DR (2009) Trasgo: a nested-parallel programming system. *J Supercomput* 58(2):226–234
10. Trasgo Group (2011) Hitmap repository. <http://trasgo.dcs.fi.uva.es/hitmap>
11. de Blas Cartón C, González-Escribano A, Llanos DR (2010) Effortless and efficient distributed data-partitioning in linear algebra. In: IEEE HPCC 2010, Sept. IEEE Press, Los Alamitos, pp 89–97
12. Fresno J, González-Escribano A, Llanos DR (2011) Automatic data partitioning applied to multigrid PDE solvers. In: 2011 19th international Euromicro conference on parallel, distributed and network-based processing. IEEE Press, Los Alamitos, pp 239–246
13. Barrett R, Berry M, Chan TF, Demmel J, Donato JM, Dongarra J, Eijkhout V, Pozo R, Romine C, Van der Vorst H (1995) Templates for the solution of linear systems: building blocks for iterative methods, 2nd edn. *Mathematics of computation*, vol 64
14. Wilkinson B, Allen M (2004) *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice Hall, New York
15. Davis TA, Hu Y (2011) The University of Florida sparse matrix collection. *ACM Trans Math Softw* 38(1):1–25
16. Bikshandi G, Guo J, Hoeflinger D, Almasi G, Fraguera BB, Garzarán MJ, Padua D, von Praun C (2006) Programming for parallelism and locality with hierarchically tiled arrays. In: PPOPP 2006. ACM Press, Los Alamitos, pp 48–57
17. Chamberlain BL, Choi S-E, Deitz SJ, Iten D, Litvinov V (2011) Authoring user-defined domain maps in Chapel. In: Cray user group