Automatic Data Partitioning Applied to Multigrid PDE Solvers

Javier Fresno
Dept. Informática
University of Valladolid
Valladolid, Spain
javier.fresno@alumnos.uva.es

Arturo González-Escribano
Dept. Informática
University of Valladolid
Valladolid, Spain
arturo@infor.uva.es

Diego R. Llanos
Dept. Informática
University of Valladolid
Valladolid, Spain
diego@infor.uva.es

***Abstract*—This paper studies the impact of using automatic data-layout techniques on the process of coding the well-known multigrid MG NAS parallel benchmark. We describe the sequential problem in detail, and discuss the parallel version and its optimizations. Then, we implement the parallel algorithm using Hitmap, a highly-efficient modular library for hierarchical tiling and mapping of arrays. We describe how to use the library plug-in system to add a new data-layout module that encapsulates a generalization of the data-alignment policy of the MG benchmark. The module system applies this policy to automatically adapt the data distribution and communication code to any grain level. The impact of using these techniques is qualitatively and quantitatively described in terms of development effort and performance. Our results show that it is possible to introduce flexible automatic data-layout techniques in current parallel compiler technology, without sacrificing performance.**

I. Introduction

Data distribution and layout is a key part for any parallel algorithm. Many problems may benefit from a multilevel data partition. Multigrid methods are a typical example. They are efficient computations of accurate numerical solutions to many scientific and engineering problems. Their parallel implementations have been widely studied (see a classical survey in [1]). To improve performance and scalability, the typical V-cycle implementations exploit different levels of parallelism. The same data partition and layout techniques are applied at different levels of the computation, with different grains. When the coarsest grids are partitioned, several issues about data location and communications arise, complicating the direct application of the same data-partition scheme.

While many languages offer a predefined set of one-level, data-parallel partition techniques (e.g. HPF [2], [3], OpenMP [4], UPC [5]), it is not straightforward to use them to create a multiple-level distributed layout that adapts automatically for different grain levels. On the other hand, message-passing interfaces for distributed environments (such as MPI [6]), or low-level thread-manipulation interfaces (such as PThreads [7]), allow to develop carefully-tuned programs for a given platform, but at the cost of a high development effort [8], [9]. The reason is that these interfaces only provide basic tools, forcing the programmer to manually code the details of data distribution and communication, with a process/thread centric view. New parallel languages propose the use of a global-view approach, with more flexible and explicit mechanisms to deal with locality (e.g. Fortress [10], Chapel [11]). A similar approach is presented in Trasgo [12], a source-to-source parallel compilation system.

The codes generated by Trasgo are supported by an extensible run-time library named Hitmap, a highly-efficient library for hierarchical tiling and mapping of arrays. It is independent enough to be used directly, as a first abstraction layer to compute, automatically map, and communicate array tiles at different levels of parallelism, greatly reducing the development effort comparing with manually-tuned implementations. Moreover, it allows the programmer to code new partitioning or data-layout techniques, and include them in the library using a system of two independent types of plug-ins: Topologies, and Partition layouts. They hide all low-level details to the programmer, avoiding the need to reason in terms of the number, or indexes, of physical processors. They are compiled externally to the library and invoked from the application code by name. This approach decouples the algorithm implementation from the data-partitioning or layout techniques selected. The programmer may test new mapping combinations of virtual topology vs. data partitioning functions, changing only the plug-in names in the whole code. This helps to implement higher levels of compiler technology to find the best mapping combination, using autotuning techniques.

In this paper we measure the impact of using automatic data-layout and communication calculation techniques to code the well-known multigrid MG NAS parallel benchmark. We analyze the reference implementation, and compare it with an alternative Hitmap code. MG is a particularly good example for this purpose, because it presents differences on the application of the data-layout on the coarser levels of parallelism, and its specific data alignment scheme needs to be coded in a new plug-in. Our experimental results show that the implementation of MG, using Hitmap, offers the same performance as the NAS implementation with a significant lower development effort. Moreover, this study shows that the Hitmap library may achieve similar performance results as sophisticated parallel languages and compilers (see comparative study in [13]), but with a higher level of flexibility to develop and apply new data layouts. Thus, it is a good candidate to be a lower layer for more complex or sophisticated parallel compiler technologies.

This paper is structured as follows: Section II describes the main features of the Hitmap library. Section III reviews the MG benchmark algorithm, including the "paper and pencil" specification, the parallel implementation details, and discusses several optimizations. Section IV describes several different MG implementations, including the NAS Fortran implementations, a new C version, and the Hitmap implementation. Section V shows experimental results concerning coding complexity and performance for the different MG implementations. Finally, Section VII concludes the paper.

II. The Hitmap library

Hitmap is a highly-efficient library for hierarchical tiling and mapping of arrays. It is designed to simplify the use of a global or local view of the parallel computation, allowing the creation, manipulation, distribution and efficient communication of tiles and tile hierarchies. In Hitmap, data-layout and load-balancing techniques are independent modules that belong to the plug-in system. The techniques are invoked from the code and applied at run-time when needed, using internal information of the target system topology to distribute the data. The programmer does not need to reason in terms of the number of physical processors. Instead, it uses highly abstract communication patterns for the distributed tiles at any grain level. Thus, coding, and debugging operations with entire data structures are easy.

The Hitmap library supports functionalities to: (1) Generate a virtual topology structure; (2) mapping the data grids to the different processor with chosen load-balancing techniques; (3) automatically determine inactive processors at any stage of the computation; (4) identify the neighbor processors to use in communications; and (5) build communication patterns to be reused across algorithm iterations.

Hitmap is designed with an object-oriented approach, although it is implemented in C language. It defines several classes, implemented as C structures and functions. It uses a compact representation for contiguous or stride domains of indexes, similar to Fortran90 or MATLAB, generalizing the contiguous rectangular regions of tools like KeLP [14]. A *Signature* is a tuple of three numbers (starting index, ending index, and stride) representing a non-contiguous but regular subset of indexes in a one-dimensional domain. A *Shape* is a set of n signatures representing a selection of indexes in a multidimensional domain. A *Tile* is an array whose indexes are defined by a shape. Tiles may be declared directly, or as a selection of another tile. Thus, they may be created hierarchically or recursively. Tiles have two coordinate systems to access their elements: Array coordinates and tile coordinates. Array coordinates associate elements with their original domain indexes, as defined by the selection shapes. Tile coordinates renumber the indexes to start always at 0 with no stride. In Hitmap, a tile may be defined with or without allocated memory. This allows to declare and to partition arrays before assigning memory to them, finally allocating only the parts mapped to a given processor.

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

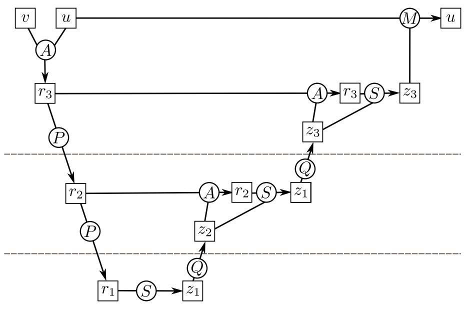


Figure 1: MG algorithm and multigrid operations.

The library provides a *Topology* plug-in class to encapsulate simple topology building functions. These modules allow to create virtual topologies from physical topology information queried internally at initialization time. Finally, the *Layout* plug-in modules allow to compute a partition of a shape domain over a virtual topology. There are some predefined layout modules in the library, but it allows to easily create new ones. The resulting layout objects have information about the local part of the input domain, neighbor relationships, and methods to get information about the parts mapped to other processes. Thus, they encapsulate data- or task-parallel mapping decisions for load-balancing and neighborhood information. The information in the layout objects may be exploited in several abstract communication functionalities, which may be composed in reusable patterns. The library is built on top of the MPI communication library, for portable communication and synchronization on different architectures. It internally exploits several MPI techniques that increase performance.

III. MG Benchmark

Regarding the benchmark used, the NAS MG Benchmark is a structured multigrid kernel to solve a discrete Poisson problem. MG is part of the NASA Advanced Supercomputing (NAS) benchmarks, developed to measure the performance of parallel computers [15]. The NAS Parallel benchmarks are well-known and widely-used applications to compare performance for different compilation techniques, communication libraries, implementations, or architectures.

A. Sequential algorithm

The NAS MG benchmark performs iterations of a V-cycle multigrid algorithm to solve a discrete Poisson problem

$\nabla^2 u = v$ on a cubical grid with periodic boundary conditions [16]. Each iteration consists of evaluation of the residual (r), and the application of the correction:

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

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

In Fig. 1 we show the algorithm for a given level, and a representation of how the operators are linked through the multigrid levels. The four operators represented in the figure are: P (Restriction), Q (Prolongation), A (Evaluation), and S (Smoother). on one dimension [16]. All these operators are easily implemented as 27 coefficients arranged as a $3 \times 3 \times 3$ cubical array.

B. Parallel algorithm

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

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

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

Figure 2 shows the application of the Q operator on a 1D grid layout at different levels. In the left side of the figure, we can see the layout for a sequential multigrid with 3 levels. In the right side, we can see the distribution in blocks. The gray squares are the ghost elements. After calculating the residual, smoother, or restriction operations, each process needs to rebuild the ghost zone with the values updated in another processor. Thus, they need to exchange their border elements with the neighboring processors.

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

IV. Implementations

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

A. NAS implementation

The NAS MG reference code uses some implementation and optimization techniques which are worthwhile to describe.

1) Memory usage: All the coarse grids for each level are allocated using a single, big array. It uses index operations based on the level of the grid and the grid position to locate and access data.

2) ZU optimization: To reduce memory usage and the number of operations of the computation, the MG reference code includes an optimization which we named *ZU optimization*. The u array is overlapped with the z array at the top level ($z_{k_{max}}$). The default operations that execute the prolongation at the top level are shown below, together with the application of the correction as it is defined by the algorithm.

$$\begin{aligned} z_k &= Qz_{k-1} && \text{(Prolongate from } k-1 \text{ to } k \text{)} \\ r_k &= r_k - Az_k && \text{(Valuate residual)} \\ z_k &= z_k + Sr_k && \text{(Prolongate)} \\ u &= u + z_k && \text{(Apply correction)} \end{aligned}$$

The optimization substitutes these operations by:

$$\begin{aligned} z_u &= z_u + Qz_{k-1} \\ r_u &= v - Az_u \\ z_u &= z_u + Sr_u \end{aligned}$$

Since the u array is set to 0 at the beginning, the sum operation can reuse values obtained previously.

3) Subexpression cache optimization: When applying the 27-point stencil of an operator, part of the partial sums computed for one element is also needed for its neighbors. The NAS implementation reduces a great number of redundant operations by caching subexpressions (partial sums) which are reused when computing subsequent elements.

4) Null coefficients optimization: The Evaluate and Smoother operators have some null coefficients. One of the simplest NAS optimizations is to skip the calculation of the contribution of these coefficients.

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

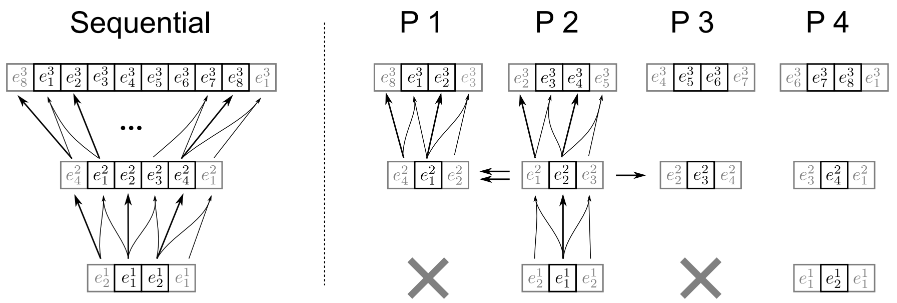


Figure 2: Application of Q operator for a partition on a given grid dimension.

B. Efficient C implementation

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

C. Hitmap Implementation

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

In Fig. 3 we show an excerpt of the Hitmap code executed at the start of the application to precompute the data layout and communication patterns. The first two lines are executed only once, to create a 3-D virtual topology of processors, using the internal information of the real topology obtained by the low-level layer (in MPI, the local identifier and the whole number of physical processors).

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

In line 14 the domain shape of the local block is obtained from the layout object, and expanded by two elements on every dimension to contain the ghost zones. This shape is used to declare and allocate the local block as a tile of double elements (line 18). All the communications needed for border exchange, and inactive processor reactivation are encapsulated using Hitmap communication patterns. In Fig. 3, line 22, a new empty communication pattern is defined, whose communication actions will be executed in order. Finally, lines 25 to 43 add communication actions to the pattern for this local processor. For each dimension, we determine the neighbors and create a shape to select which part of the tile is sent or received. Since the layout object contains all the information about shapes, active processors, and neighborhoods, the code between lines 25 to 43 are common to both, active and inactive processors, at any level of the multigrid computation. Data marshaling and unmarshaling is automatized by the actions of the communication patterns. When a processor needs to exchange the borders, it simply executes the appropriate pattern for the level with a single Hitmap function call whose only parameter is the pattern (not shown in the figure). The patterns are reusable through all the algorithm iterations.

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

D. Building a layout plug-in

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

Hitmap have two types of data-layout modules with a similar interface. The simplest one calculates appropriate slicings of index domains which may be expressed as a signature, like blockings, or cyclic. The current version of the Hitmap library provides two different blocking modules: *Blocks*, and *BlocksF*. Both modules balance the partition in the same way when there are more domain elements than processors on a given dimension. However, when there are not enough domain elements, *Blocks* concentrates the domain elements on the lower processor indexes of the dimension, while *BlocksF*

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

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

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

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

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

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

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

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

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

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

Figure 3: Construction of the data layout and communication patterns for MG.

creates even groups of processors and assigns each domain element to the First processor of each group. The other processors are marked as inactive on the layout object.

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

Instead of modifying the data-alignment and the indexes used to apply operators in the application code, it is easier to build a new layout plug-in for the library. We named it `BlocksL` (Blocks with active leader at the Last processor of the group). A new plug-in is defined by two functions, one to calculate the proper slicing, and another to define the neighborhood relationship (skipping possible inactive processors), for one dimension domain. The plug-in system systematically applies these functions to the required dimensions.

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

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

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

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

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

$$s' = s$$

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

More sophisticated partition modules may define extra parameters. For example, some modules receive an estimation

```
int hit_layout_plug_layBlocksL_Sig(int procId, int procsCard, HitSig in, HitSig *out ) {
      int blocksCard = hit_sigCard( in );
      double ratio = (double)blocksCard / procsCard;
      double beginFrac = procId * ratio;

      /* DETECT NON-ACTIVE VIRTUAL PROCESS (NOT ENOUGH LOGICAL PROCESSES) */
      if ( floor(beginFrac) == floor(beginFrac+ratio) ) {
            (*out) = HIT_SIG_NULL;
            return FALSE;
      }
      else {
            /* COMPUTE SIGNATURE */
            (*out).begin = (int)beginFrac * in.stride + in.begin;
            int adjust = (blocksCard > procsCard) ? 1 : 0;
            (*out).end = (((procId + adjust) * (int)ratio ) - adjust) * in.stride + in.begin;
            (*out).stride = in.stride;
            return TRUE;
      }
}
```

Figure 4: Plug-in function to generate a blocking partition compatible with MG, for any dimension.

```
int hit_layout_plug_layBlocksL_neighbor(int procId, int procsCard, int blocksCard, int shift, int wrap) {
      /* COMPUTE ACTIVE NEIGHBOR */
      double ratio = (double) blocksCard / procsCard;
      if(ratio > 1) ratio = 1;
      int neighIdAct = (int)(procId * ratio) + shift;
      int activeProcs = min(procsCard,blocksCard);

      /* CHECK IF THE NEIGHBOR IS OUT OF THE PROCESSORS RANGE: APPLY WRAPPING */
      if ( neighIdAct < 0 || neighIdAct >= activeProcs )
            if(wrap != HIT_WRAPPED) return HIT_RANK_NULL;
            else neighIdAct = ((neighIdAct % activeProcs) + activeProcs) % activeProcs;

      /* RETURN THE LAST PROCESSOR OF THE GROUP (IT IS LOCATED BEFORE THE NEXT GROUP) */
      return (int) ceil((neighIdAct+1) * 1/ratio) -1;
}
```

Figure 5: Plug-in function to handle neighborhood relationships compatible with MG.

of the load associated with the domain indexes, which may be used to generate an adaptive load-balance, or to decline further parallelization when the grain is too fine.

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

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

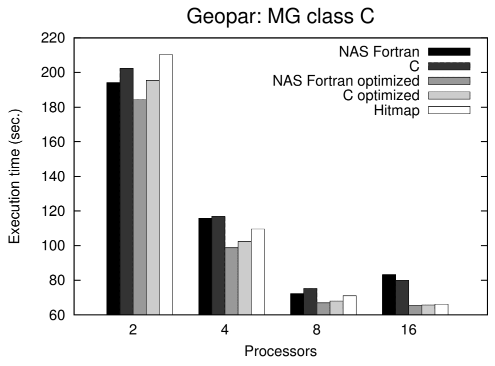
Testing combinations of topology vs. layout functions is really easy, changing only the names or parameters of the topology or layout functions. For example, we may use a *mesh1D* topology plug-in to create a one dimensional collection of coarse 3D blocks. Or use an extra parameter to restrict the layout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

V. Experimental results

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

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

We also show the performance of two additional versions: "NAS Fortran Optimized", consisting of the original NAS code but without the unnecessary clean of the reception buffers



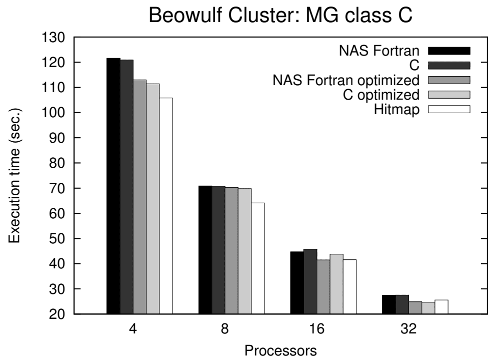


Figure 6: Comparison of execution times between NAS Fortran, C manual optimized and Hitmap versions.

(see Sec. IV-A), and "C optimized", that consists of the C version of this optimized code. These versions were developed to isolate the effect of this optimization, since Hitmap does not need to clean reception buffers.

Figure 6 shows the execution time of the main computation part of these five versions. In the Geopar system all versions surpass the original NAS code. The effect of the optimized versions is noticeable in terms of performance, as expected. Hitmap uses complex macro functions to expose to the native C compiler the formula used to access tile elements. The optimizations obtained with the Solaris compiler are not as good as with GCC. Thus, the sequential time on the Geopar machine leads to slightly higher execution times of the Hitmap version for few processors. Nevertheless, Hitmap version scales better than the others due to its reduced communication costs. In the Beowulf cluster (Fig. 6, bottom), the use of Hitmap leads to even better performance figures, since communications are more costly in this architecture, and Hitmap takes advantage of several MPI advanced capabilities automatically. For example, the Hitmap communication functions precalculate hierarchical MPI derived data types for each tile. The resulting objects are efficiently reused on each iteration. The beneficial effects of using appropriate MPI derived data types for communications in terms of performance has been reported in [17].

Finally, Fig. 7 shows a comparison of the Hitmap version with the Fortran and the manually optimized C versions in terms of lines of code. We distinguish lines devoted to sequential computation, declarations, parallelism (data layouts

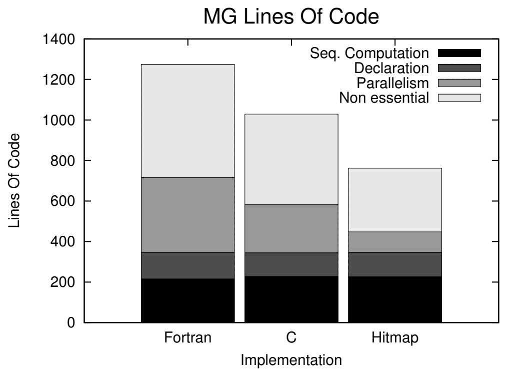


Figure 7: Comparison of the number of code lines.

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 37.4% reduction on the total number of code lines with respect to the Fortran version, and a 23.2% reduction with respect to the C version. Regarding lines devoted specifically to parallelism, the percentages of reduction are 72.7% and 57.4%, respectively.

The reason for the 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 needed to manually calculate the information needed to be used in the MPI routines. In particular, Hitmap avoids the use of tailored formula to compute local tile sizes, mapping of spare data to virtual processors, and neighborhood relationship, at the different grain levels. Moreover, the use of Hitmap tile-management primitives eliminates some more lines in the sequential treatment of matrices and blocks.

VI. Related work

Hitmap has been designed to provide an integrated, common interface to exploit several techniques related to data-partition, mapping, and communication. It introduces tools for dynamic tile creation, parametrized by the result of data-partition plug-ins, also adequate for recursive or domain decomposition techniques [18]. Hitmap provides all the tools to build more abstract data types for tiling arrays (such as HTA [19]), but introducing a new level of flexibility and extensibility on the data partition, and introducing a generalized hierarchy system, suitable for irregular or adaptive grids.

Hierarchical storage similar to Flame [20] is naturally created in Hitmap using tiles which base type is another tile. Hitmap does not encapsulate the exact layout hierarchy on the data type, but it allows to define it dynamically, or by another layout function.

The automatic generation of data communication patterns from contiguous n-dimensional overlapped regions has been previously implemented in the KeLP library [14]. The KeLP architecture, with object classes for expressing data partitions, mappings, and reusable communication patterns, is further refined in Hitmap. Hitmap transparently supports contiguous or stride domains for efficient cyclic distributions. It also proposes a common interface for regular or irregular partition techniques, effectively isolating the application code from data-partitioning decisions which imply reasoning in terms of the processor identification or the number of processors. Communication objects in Hitmap are built in terms of algorithm data dependencies and data-partition information generated automatically at run-time, generalizing the KeLP communication calculation, that is mainly based on domain overlappings. Hitmap layouts only store a fixed amount of local information, providing a more efficient and scalable implementation.

The Chapel language proposes a transparent plug-in system for domain partitions, although no complete specification, implementation, nor experimental results are available yet [21]. Chapel proposes only one type of partition plug-ins, eliminating the flexibility introduced by the topology and layout combinations of Hitmap. It also forces to create new specific modules for partitions which may be expressed by multilevel combinations of layouts in Hitmap (such as block-cyclic).

VII. Conclusions

This paper studies the impact of using automatic data-layout techniques on the process of coding the well-known multigrid MG NAS parallel benchmark. We have implemented the parallel algorithm with Hitmap, a highly-efficient modular library for hierarchical tiling and mapping of arrays. The impact of using the library is qualitatively and quantitatively described in terms of development effort and performance. We have also described how to use the plug-in system of the library to add a new data-layout module, with a generalization of the data-alignment policy of the MG benchmark, in order to automatically adapt the data-distribution and communication code to any grain level. We show how the Hitmap approach can be used to generalize and encapsulate data-partition policies, providing a higher degree of application control and flexibility due to its decoupled mapping system, based on two types of combinable plug-ins. Our results show that it is possible to introduce flexible automatic data-layout techniques in current parallel compiler technology, greatly reducing the development effort when comparing with coding these details manually, without sacrificing performance.

We are currently working on extending the HitShape objects to support more irregular domains, such as sparse matrices, or graphs. A new family of applications and partition techniques, currently isolated on different tools, could be integrated in the Hitmap framework. Also, we are working on other implementation alternatives that exploit different low-level parallel tools and models than the MPI message-passing interface, in order to generalize these results for heterogeneous environments.

Acknowledgements

This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302), Ministerio de Industria, Spain (TSI-020302-2008-89, CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER), Junta de Castilla y León, Spain (VA094A08), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research.

References

[1] J. Jones, "Parallel multigrid methods," in *Parallel Numerical Algorithms*, D. K. et al., Ed. Kluwer Academic Publishers, 1997, pp. 203–224.

[2] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of High Performance Fortran," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III*, San Diego, California, 2007, pp. 7.1–7.22.

[3] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD distributed-memory machines," *Commun. ACM*, vol. 35, no. 8, pp. 66–80, 1992.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*, 1st ed. Morgan Kaufmann Publishers, 2001, iSBN 1-55860-671-8.

[5] 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.

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

[7] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[8] S. Gorlatch, "Send-Recv considered harmful? Myths and truths about parallel programming," in *PaCT'2001*, ser. LNCS, V. Malyshkin, Ed., vol. 2127. Springer-Verlag, 2001, pp. 243–257.

[9] K. Gatlin, "Trials and tribulations of debugging concurrency," *ACM Queue*, vol. 2, no. 7, pp. 67–73, Oct 2004.

[10] G. Steele, "Parallel programming and code selection in Fortress," in *PPoPP'06*. ACM Press, March 2006, pp. 1–1.

[11] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, Aug 2007.

[12] A. Gonzalez-Escribano and D. Llanos, "Trasgo: A nested parallel programming system," *Journal of Supercomputing*, vol. doi:10.1007/s11227-009-0367-5, 2009.

[13] B. L. Chamberlain, S. J. Deitz, and L. Snyder, "A comparative study of the NAS MG benchmark across parallel languages and architectures," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Dallas, Texas, United States: IEEE Computer Society, 2000, p. 46.

[14] S. Baden and S. Fink, "The Data Mover: A machine-independent abstraction for managing customized data motion," in *Languages and Compilers for Parallel Computing, LCPC'99*, ser. LNCS, vol. 1863. Springer, 2000, pp. 333–349.

[15] D. Bailey, T. Harris, W. Saphir, R. van der Winjgaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," NASA Advanced Supercomputing (NAS) Division, Report RNR-95-020, 1995.

[16] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," NASA Advanced Supercomputing (NAS) Division, Report RNR-94-007, 1994.

[17] Q. Lu, J. Wu, D. Panda, and P. Sadayappan, "Applying MPI derived datatypes to the NAS benchmarks: A case study," in *2004 International Conference on Parallel Processing Workshops (ICPPW'04)*. IEEE, 2004, pp. 538–545.

[18] Q. Yi, V. Adve, and K. Kennedy, "Transforming loops to recursion for multi-level memory hierarchies," in *Proceedings of the ACM SIGPLAN PLDI*. Vancouver, British Columbia, Canada: ACM, 2000, pp. 169–181.

[19] 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.

[20] 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.

[21] 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.