Parallel Computing 38 (2012) 465–484

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Optimization techniques for efficient HTA programs

Basilio B. Fraguela [a,*], Ganesh Bikshandi [b], Jia Guo [c], María J. Garzarán [c], David Padua [c], Christoph von Praun [d]

[a] Depto. de Electrónica e Sistemas, Universidade da Coruña, Facultade de Informática, Campus de Elviña, S/N, 15071 A Coruña, Spain
[b] Intel Labs, Intel Technology India Pvt. Ltd., Bangalore 560 103, Karnataka, India
[c] Dept. of Computer Science, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, 61801 IL, USA
[d] Fakultät Informatik, Georg-Simon-Ohm Hochschule, Postfach 210320, 90121 Nuremberg, Germany

ARTICLE INFO

Article history:
Received 20 May 2011
Received in revised form 11 May 2012
Accepted 14 May 2012
Available online 22 May 2012

Keywords:
Parallel programming
Optimization
Programmability
Libraries
Data-parallel
Tiling
Locality
Runtime overheads

ABSTRACT

Object oriented languages can be easily extended with new data types, which facilitate prototyping new language extensions. A very challenging problem is the development of data types encapsulating data parallel operations, which could improve parallel programming productivity. However, the use of class libraries to implement data types, particularly when they encapsulate parallelism, comes at the expense of performance overhead.

This paper describes our experience with the implementation of a C++ data type called hierarchically tiled array (HTA). This object includes data parallel operations and allows the manipulation of tiles to facilitate developing efficient parallel codes and codes with high degree of locality. The initial performance of the HTA programs we wrote was lower than that of their conventional MPI-based counterparts. The overhead was due to factors such as the creation of temporary HTAs and the inability of the compiler to properly inline index computations, among others. We describe the performance problems and the optimizations applied to overcome them as well as their impact on programmability. After the optimization process, our HTA-based implementations run only slightly slower than the MPI-based codes while having much better programmability metrics.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Parallelism can be introduced with new languages, language extensions, compiler directives or libraries. The main benefit of using libraries instead of compilers is that libraries are typically easier to implement and port. Their main drawbacks are verbosity and their potential performance overhead. Much of the verbosity can be avoided if the libraries are implemented in object oriented languages with polymorphism and operator overloading. Measuring, characterizing and proposing techniques to mitigate this overhead is an essential task to explore the usage of libraries as a vehicle for bringing parallelism into applications. In this paper we discuss our efforts to address the overhead of the initial implementation in C++ of the Hierarchically Tiled Array (HTA) [1,2], a class that allows direct control of locality and parallelism by means of tiles. HTAs have three important features that facilitate parallel programming (1) they provide a single-threaded view of the parallel execution by following a data-parallel approach (2) present the programmer with a global view of distributed data and (3) they serve as a common mechanism to represent parallel operations across classes of parallel machines: shared-memory multiprocessors, multicomputers, and SIMD processors. While the impact of HTAs to ease parallel programming has been

* Corresponding author. Tel.: +34 981 167000x1219; fax: +34 981 16 71 60.
E-mail addresses: basilio.fraguela@udc.es (B.B. Fraguela), ganesh.bikshandi@intel.com (G. Bikshandi), jiaguo@uiuc.edu (J. Guo), garzaran@illinois.edu (M.J. Garzarán), padua@illinois.edu (D. Padua), praun@ohm-hochschule.de (C. von Praun).

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.05.002

discussed in previous publications, a study of performance and the techniques needed to improve it as well as their impact on programmability is presented for the first time in this paper.

The rest of this paper is organized as follows. Section 2 introduces HTAs. Then, Section 3 discusses implementation details of the class, including the optimizations that could be implemented in it and which are completely transparent to the programmer. Section 4 discusses the optimizations that the programmer has to manually apply in the programs to achieve high performance. The applications used for the evaluation (NAS benchmarks [3]) and the results obtained are discussed in Section 5, followed by related work in Section 6. Finally Section 7 presents our conclusions.

2. Hierarchically tiled arrays

This section outlines the semantics of the hierarchically tiled arrays (HTA) (Section 2.1), their construction (Section 2.2), access mechanisms (Section 2.3), assignment statements and binary operations (Section 2.4) and HTA methods (Section 2.5). HTA usage is also illustrated with small codes in Section 2.6. More details can be found in [1,4,2]. We use a MATLAB-like syntax.

2.1. Semantics

Hierarchically tiled arrays (HTAs) are arrays partitioned into tiles. These tiles can be either conventional arrays or lower level hierarchically tiled arrays. Tiles can be distributed across processors in a distributed-memory machine or be stored in a single machine according to a user specified layout. In distributed-memory machines the outermost tiles are often distributed across processors for parallelism and the inner tiles are used to improve locality within a processor. Fig. 2(a) shows an example HTA with two levels of tiling.

2.2. Construction of HTAs

HTAs are typically created by providing the number of tiles per dimension, and the size of each tile. Fig. 1(a) defines a 3×3 HTA H with tiles of size 2×2. In general, one can create an HTA with l levels of tiling, using the HTA constructor $H = hta(\{[s_0, s_1, \ldots, s_{d-1}]^0, \ldots, [s_0, s_1, \ldots, s_{d-1}]^l\})$, where d is the number of dimensions of the HTA, and each argument i, where $i \in \{0 \ldots l\}$ specifies the dimensions of the HTAs at level i.

The tiles of an HTA can be local or distributed across processors. To map tiles to processors, the topology of the mesh of processors and the type of distribution (block, cyclic, block cyclic, or a user-defined distribution) must be provided. Fig. 1(b) shows an example where a 6×6 matrix is distributed on a 2×2 mesh of processors. The last parameter of the HTA constructor specifies the processor topology. In the current implementation, the default distribution is block cyclic.

2.3. Accessing the components of an HTA

Fig. 2(b) shows examples of how to access HTA components. The expression `C{2,1}` refers to the lower left tile. The scalar element in the fifth row and fourth column can be referenced as `C(5,4)` just as if `C` were an unpartitioned array. This element can also be accessed by selecting the bottom-level tile that contains it and its relative position within this tile: `C{2,1}{1,2}(1,2)`. A third expression representing `C(5,4)` selects the top-level tile `C{2,1}` that contains the element and then *flattens* or disregards its internal tiled structure: `C{2,1}(1,4)`. Flattening is particularly useful when transforming a sequential program onto a tiled form for locality/parallelism or both. During the intermediate steps of the transformation, some regions of the program can remain unmodified because conventional array accesses always have the same semantics even when the array has been transformed into HTA.

In any kind of indexing, a range of components may be chosen in each dimension using triplets of the form *begin*:*step*:*end*, where *begin*, *step* and *end* are optional. The default values when none is provided are the first element for *begin*, 1 for *step*, and the last element for *end*. Also, the : notation can be used in any index to refer to the whole range of possible values for that index. For example, `C{2,:}(1:2:4,:)` refers to the the odd rows of the two lower outer-level tiles of `C`.

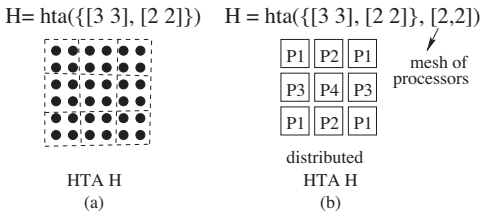


Fig. 1. Construction of an HTA by partitioning an array-(a). Mapping of tiles to processors-(b).

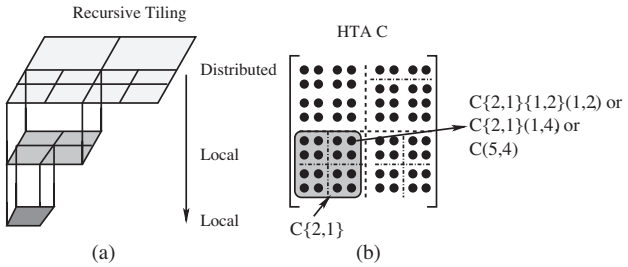


Fig. 2. Pictorial view of a hierarchically tiled array.

2.4. Assignments and binary operations

HTAs generalize the notion of conformability of Fortran 90. When two HTAs are used in an expression, they must be conformable. Specifically, they are conformable if they have the same topology (number of levels and shape of each level), and the corresponding tiles in the topology must have sizes that allow to operate them. The operation is executed tile by tile, and the output HTA has the same topology as the operands.

Also, an HTA can be conformable to an untiled array and it is always conformable to a scalar. In the first case, the array must be conformable with each one of the innermost tiles of the HTA. When an untiled array is operated with an HTA, each leaf tile of the HTA is operated with the array. Also, when one of the operands is a scalar, it is operated with each scalar component of the HTA. Again, the output HTA has the same topology as the input HTA.

Assignments to HTAs are governed by the same rules of binary operators. When a scalar is assigned to a range of positions within an HTA, the scalar is replicated in all of them. When an array is assigned to a range of tiles of an HTA, the array is replicated to create tiles. Finally, an HTA can be assigned to another HTA (or a range of tiles of it).

In a distributed memory machine references to local HTAs do not involve communication. However, distributed HTAs have their outer tiles distributed on a mesh of processors and assignments between tiles which are in different processors involve communication. Consider a distributed HTA `V` of 3×3 tiles of $n \times n$ elements. The assignment `V{2:3,:}(1,:) = V{1:2,:}(n,:)` copies all the elements in the last row of the rows 1:2 of tiles to the first row in the rows 2:3 of tiles as shown in Fig. 3. When the tiles of `V` are distributed across processors, this assignment involves communication.

2.5. Methods

Table 1 lists the main HTA operations, categorized as point-wise, collective, higher-order or dynamic partitioning. The point-wise operations include the standard arithmetic operations, such as addition, that affect each of the scalar values of an HTA, but are applied at the tile level when applied to an HTA. Point-wise operators are classified as unary or binary, based on the number of arguments; assignment operations also belong to this category. These operations need to follow the conformability rules that have been described in the previous section.

Collective operations are those that do not change the values of the scalar elements of the HTA, but their positions in the HTA. Thus, the output HTA of these operations often does not have the same structure as the input HTA. This is the case of the methods that permute or transpose the elements in an HTA. For example, Fig. 4 illustrates the difference between methods `permute` and `dpermute`. While the former permutes the dimensions of an HTA at every tiling level, exchanging the dimensions specified as input, `dpermute` permutes only the data in an HTA, without changing the tiling structure. `dpermute` achieves this by working on an HTA with N top level tiles that are subtiled in N two-level tiles ($N = 2$ in Fig. 4) and copying a permuted version of tile `{i}{j}` of its input HTA to tile `{j}{i}` of the output. Another example of collective method is `repmat`, which replicates the tiles of an HTA. For the `repmat` method the programmer provides as input a vector with as many elements as dimensions the HTA has, where the ith element indicates the number of replicas of the HTA for the ith dimension.

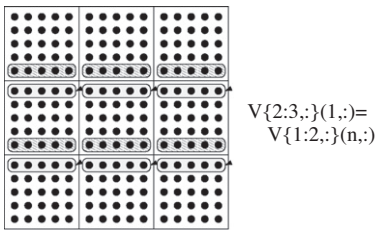


Fig. 3. Assignment of the elements in the last row of the tiles in rows `1:2` to the first row in the tiles in the rows `2:3`.

Table 1
Summary of HTA operations.

Class	Type	Shape of output	Input operator (if any)
Point-wise	Unary	Shape is unchanged	Nil
	Binary	Depends on the inputs	Nil
	Assignment	Shape is unchanged	Nil
Collective	`permute`	Permuted shape of the input HTA	Permutation of the dimensions
	`dpermute`	Shape is unchanged	Permutation of the dimensions
	`transpose`	Transposed 2D input HTA	Nil
	`htranspose`	Output tile {i}{j} ⇐ input tile {j}{i}	Nil
	`repmat`	Depends on the dimension	Replications per dimension
Higher-order	`reduce`	depends on the dimension and level	Associative operation and dimension
	`scan`	Depends on the dimension and level	Associative operation and dimension
	`hmap`	Shape is unchanged	Any scalar, array or HTA operation
Dynamic partitioning	`part`	Change tiling structure	Partition and offset
	`rmPart`	Change tiling structure	Partition

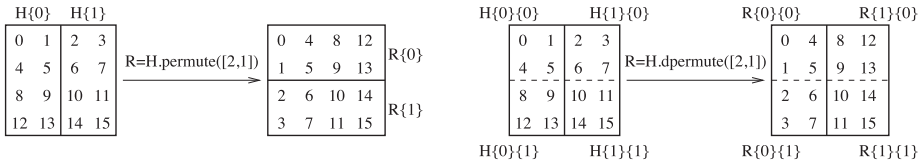


Fig. 4. HTA `permute` versus `dpermute`.

Higher order operators are parametrized with primitive operators and define the strategy and result format resulting from the application of their input operator to the tiles or scalar values of an HTA or of several HTAs in the case of `hmap`. This way, `reduce` is a generalized reduction method that operates on HTA tiles, where the user needs to specify the reduction operation, an associative operation such as addition. For HTAs with more than one dimension, the programmer can specify optional parameters such as along which dimension the reduction needs to be performed. By default the reduction is performed all the way to the scalar level, but the programmer can use an optional parameter to indicate the level where the recursion level of the reduction needs to stop. The `scan` method computes the reductions of all the prefixes of an HTA, and uses similar parameters to `reduce`. Finally, `hmap` applies in parallel the same function to each tile of an HTA, or to the corresponding tiles of different HTAs when it is applied to more than one HTA.

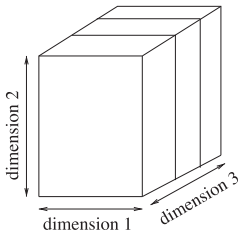
Another feature of HTAs is dynamic partitioning [2], that can be used to modify the HTA tiling structure on the fly. Dynamic partitioning is based on *partition lines*, the lines that separate tiles in an HTA. These lines are numbered starting with 0 in each dimension, with line 0 being the one that implicitly exists before the first element in a dimension. A *partition* is a set of partition lines with 0 or 1 lines per dimension. It is represented by a tuple that for each dimension contains a partition line number or the special value `NONE` when there is no partition line for a given dimension. We provide the methods `part` operation to add a partition and `rmPart` to remove a partition.

Although the next section contains several examples illustrating the usage of HTAs, the reader can refer to [5] for a more detailed explanation of the HTA operations, including syntax and examples. Dynamic partitioning is described in [2,6].

2.6. Examples of usage

NAS benchmark FT operates on a 3-D array that is partitioned into tiles which are distributed along the third dimension, as shown in Fig. 5(a). To compute the Fourier Transform (FT) of a 3-D array, FT needs to be applied along each of the dimensions. However, to perform the FT along the third dimension the blocks from the distributed dimension need to be brought from the distributed dimension to an undistributed one, so that FT can be locally applied. Fig. 5-(b) shows an outline of the NAS FT code using HTAs. The FT is applied along the first and the second dimension of an HTA using the `hmap` operator, which applies in parallel function `fft` to all the tiles of the HTA `X`. To apply the FT along the third dimension, we use the `dpermute` operator to make the third dimension local to a processor.

The second example is the matrix–vector multiplication in Fig. 6. This computation is the core of the NAS benchmark CG. HTA `A` is made up of $m \times n$ tiles (of $s1/m \times s2/n$ elements each) distributed on a mesh of $m \times n$ processors. HTA `V` contains the vector for the product, distributed in blocks of $s2/n$ elements only on the first row of processors of the mesh, as it has a single row of tiles. HTA `B` is obtained by replicating `V` `m` times in the dimension of the rows as specified by the operator `repmat` to create a copy on each row of processors. The matrix–vector multiplication `C = A * B` takes place locally, that is, each processor multiplies its portion of the matrix `A` by its portion of the vector in `B`. Notice that each row tile of `B` has been first




```
X=hta({[1,1,nprocs],
   [s1,s2,s3/nprocs]},
        [1,1,nprocs]);
X.hmap(fft,1);
X.hmap(fft,2);
Y=X.dpermute([3,1,2]);
Y.hmap(fft,1);
```

(b)

Fig. 5. FT benchmark kernel.

```
A = hta({[m, n], [s1/m, s2/n]}, [m, n]);
V = hta({[1, n], [1, s2/n]}, [m, n]);
B = V.repmat(m, 1);
B.hmap(transpose);
C = A * B;
C = C.reduce(plus-operator, 2, true);
```

Fig. 6. Matrix–vector multiplication example.

```
void PMerge (HTA out, HTA in1, HTA in2) {
  int in1_size = in1.size(0);
  if (in1_size > GRAINSIZE) { /* parallel merge */
    int midpos_in1 = in1_size / 2 ;
    int cutpos_in2 = in2.lowerBoundPos(in1[midpos_in1]);
    in1.part( [0], [midpos_in1] );
    in2.part( [0], [cutpos_in2] );
    out.part( [0], [midpos_in1  + cutpos_in2] );
    hmap(PMerge, out, in1, in2);
    in1.rmPart();
    in2.rmPart();
    out.rmPart();
  } else {  /* sequential merge */ }
}
hmap(PMerge, out, in1, in2);
```

Fig. 7. Parallel merge using HTA dynamic partitioning.

transposed within each processor into a column by `hmap(transpose)`. After the multiplication, a reduction along the second dimension of `C` (i.e. adding the portions of `C` in the same row of processors), as specified in the second parameter of `reduce`, generates the final result. HTA `C` is a column vector distributed across the m rows of our $m \times n$ mesh and replicated along its n columns of processors. The reason for this replication is that the usage of `true` as last parameter for `reduce` requests an all-to-all reduction.

Finally, Fig. 7 applies dynamic partitioning to merge two HTAs `in1` and `in2` containing a sequence of sorted values into a single sorted sequence in HTA `out`. The three HTAs used are unidimensional and initially have a single tile. If `in1` is larger than threshold `GRAINSIZE` the HTAs are partitioned into two tiles, so that the merging of the first tiles of `in1` and `in2` can proceed in parallel with the merging of the second tiles. Otherwise, the operation is performed sequentially. After determining where to partition `in1` and `in2`, HTA `out` is also partitioned to accommodate the result of the merge of the corresponding input tiles. The `part` method receives as parameters a partition ([0] in this case) and an offset from that partition where the new partition should be created. Function `hmap` applies a functor in parallel to the corresponding tiles of a set of HTAs. In our case, it generates two parallel invocations of the `PMerge` function: one on `out(1)`, `in1(1)` and `in2(1)`; and another one on `out(2)`, `in1(2)` and `in2(2)`. Both can run in different threads in a shared memory multiprocessor/multicore, while communication of tiles would be needed in a distributed memory environment. Function `hmap` returns when both parallel merges have finished. Then the partitions are removed from the HTAs with method `rmPart`, which can be invoked with a tuple identifying the partition to be removed or without arguments, in which case all the partitions are removed.

3. Library implementation

The C++ implementation of the HTA class is a library with ∼ 18000 lines of code, excluding comments and empty lines. It only contains header files, as most classes in the library are C++ templates to achieve generality in the data types of tiles and to enable compile-time polymorphism [7]. The library has been designed in a modular way to allow flexibility in the choice of the underlying communication and threading mechanisms. There are a series of classes implementing abstract functionalities on the data type, some of which must be specialized to enable execution on an specific runtime and communication system. The stable implementation has sequential, multi-threaded (SMP) and distributed runtime systems, the two latter ones being implemented on top of Intel Threading Building Blocks [8] and MPI [9], respectively. The runtime system is selected at compile time by means of a user-defined constant. Programs that use the HTA library do not need to be modified when using a different runtime system or machine class [10].

In this paper, we focus on the experimental results obtained when using the MPI-based runtime system, although many of the issues affect all of them. For the MPI-based one the execution model chosen is the Thread Private or Single Program Multiple Data (SPMD). Shared data such as scalars, arrays and non-distributed HTAs are replicated across all processors. Information about distributed HTAs is created in all the processors, so that they all know the structure of the HTAs. However, processors only store the data of the tiles in the distributed HTAs that they own. Every processor executes the whole program. When operating on distributed HTAs, each processor applies locally the operation on the tiles of the HTA it owns. In this case explicit synchronization between processors is not necessary. Our run time library uses a two-sided communication model where the synchronization is always implicit because communication takes place from the producer to the consumer, so an eager consumer always has to wait for a delayed producer to send the data before it can proceed with the computation. Thus, it is not necessary to synchronize the processors before or after a parallel computation; synchronization will take place on demand when data from other processor is needed. With this execution model it is possible to execute two or more statements concurrently, without affecting the sequential deterministic semantics. Consider the example below where `H` is the HTA of Fig. 1 distributed using block-cyclic distribution, as shown in Fig. 1. Thus, tiles {1,1}, {1,2}, {2,1} and {2,2} are owned by processors 1, 2, 3 and 4 respectively, and `mtrx1` and `mtrx2` are two matrices.

```
(1) H{1:2, 1} = mtrx1;
(2) H{1:2, 2} = mtrx2;
```

All the processors try to execute statement (1) but since only processors 1 and 2 own the tiles involved in this assignment, processors 3 and 4 can skip it and proceed to execute statement 2, while processors 1 and 2 perform the assignment. In this second statement, processors 3 and 4 are the ones that will perform the assignment. Thus, in this case, statement 1 and 2 execute concurrently on different processors.

The rest of this section describes some of the potential overheads in a library implementation for the HTA and discusses the approach taken to optimize the library and reduce these overheads in a way transparent to the user. That is, the optimizations described next are provided automatically by our library to any HTA program, requiring no effort from the programmer.

3.1. Avoidance of creation of temporary HTAs in the library

In expressions operating with HTAs such as `D = A + B + C` a function call is performed on each overloaded arithmetic operation (that returns a temporary HTA) and assignment operation. The reason is that HTA is a C++ class and "+" is just a method. Thus, the previous expression represents three function calls to the `hmap` function, as seen in Fig. 8(a). The `hmap` operator applies the function specified in the first parameter to the input HTAs. Thus, the expression `D = A + B + C` is in effect implemented as the three loops in Fig. 8(b), where the reference `x{i}` stands for a complete tile. Unfortunately, this implementation can degrade performance significantly due to the overheads of creating temporary HTAs, the loop overheads and the additional cache and TLB misses.

This problem can be solved by using expression templates [11] delaying the evaluation of the right hand side of the expression until the target of the assignment is determined. This way, the three previous loops are fused in a single loop `D{i} = A{i} + B{i} + C{i}`. Our current implementation only addresses the case where there are two operands on the right hand side. The reason is that this design point maximizes the balance between the performance improvement achieved

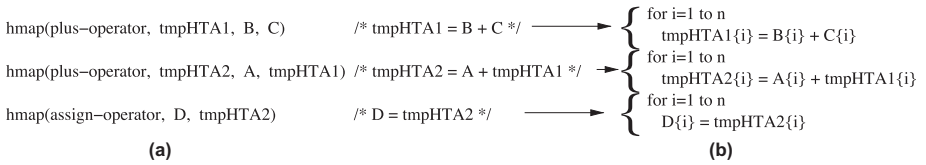


Fig. 8. Evaluation of HTA expression `D = A + B+C` without optimizations.

and the complexity introduced in the library. Thus, `C = A + B` is fully optimized, but the example above with three operands results in two loops, one that adds `B` and `C` in a temporary, and another one that adds `A` and the temporary HTA into `D`. When necessary, the user can avoid manually the overheads due to temporaries using the solution presented in Section 4.1.

We have also used expression templates to eliminate temporaries in other situations, such as assignments of the kind `B = OP(A)`. The execution of `OP` is delayed until the point of the assignment. Then the result is generated directly in the destination instead of in a temporary HTA which is to be copied to the destination.

Another case when temporary HTAs could appear is when indexing some tiles of an already existing HTA. To avoid them, this library does not create deep copies of the data; instead it creates structures and masks that point to the original data.

3.2. Reuse of dead HTAs

The cost of creating new HTAs is high. Thus, our library tries to reuse previously created HTAs when a new HTA is necessary. Our implementation contains a pool of HTAs that went out of scope, i.e., which are no longer in use. An HTA from this pool is recycled when it is conformable (has same dimensions and size in each dimension) with the one that needs to be created. The size of the pool can be adjusted dynamically as memory becomes available. It can also release the space of those HTAs that are found not to be used frequently.

3.3. HPF-like optimizations

Tile ownership must be examined by each parallel processor or thread to (a) determine which tiles it is responsible for and thus operate on them (owner's compute rule) and (b) know with which processor to communicate in distributed memory environments. This test process is more costly the more tiles the HTA has, particularly because the initial implementation used an array to map each tile to its owner. To reduce overheads, we first moved to the usage of symbolic formulae for the canonical distributions such as cyclic or block cyclic. These canonical mappings suffice for all the applications we studied. Our second improvement was to add a vector to the representation of the HTA in each processor that stores the indexes of the owned tiles. This vector is used by operations that only work on local tiles, such as `hmap`, in order to iterate directly only on those tiles instead of inspecting the ownership of every tile.

4. User level optimizations

The optimizations applied inside the library play a very important role in its performance. However, they are insufficient for the HTA version of the NAS benchmarks to perform as well as their FORTRAN + MPI counterpart. Below we describe some of the optimizations that the programmer has to apply when writing the applications including an estimation of their programmability cost in SLOCs (Source Lines Of Code).

4.1. Avoidance of creation of temporaries in the user code

4.1.1. Temporaries resulting from intermediate computations

As discussed in Section 3.1, our library automatically eliminates temporaries in operations with up to two operands on the right hand side. For those cases where more than two operands are required, we used the `hmap` function described in Section 2.5. This way, our example `D = A + B + C` is actually written as `hmap (f, D, A, B, C)`, where `f` is a function or C++ functor that in each invocation receives as input a corresponding tile of its input HTAs and performs the computation with a loop, avoiding the generation of temporaries. A typical implementation looks like the following:

```
function f (HTA D, HTA A, HTA B, HTA C) {
  for i = 1 to size(D)
    D(i) = A(i) + B(i) + C(i);}
```

As we see this technique requires that the output HTA is built in advance, which implies that the user must be able to infer its dimensions and tiling. Also, due to the semantics of `hmap`, the computation must be applicable at tile level, on the corresponding tiles of the input HTAs, and it must not involve communications.

This optimization is very important for scientific codes, as expressions with multiple operators appear in many computations. This way, other libraries [12,13] that provide array-based expressions have also had to cope with this problem, usually resorting to expression templates [11]. Still, this technique introduces other problems, like obfuscating the code presented to the compiler, and thus potentially missing optimization opportunities [14]. For this reason these libraries add special classes, like stencils, which optimize typical computational patterns inside the library. We have preferred a more general solution which allows the user to perform further optimizations that will be discussed in Section 4.2.

Compilers for array-based languages have to deal with this problem too, as intermediate arrays are introduced both at the source level and during the compilation process. Most compilers have addressed the problem by scalarizing [15] the array language before performing loop fusion, which requires additional considerations in distributed-memory environments [16],

followed by the removal of unneeded temporary arrays through array contraction [17]. Some compilers [18] perform array-level analysis to determine the best scalarization and fusion strategy for the purpose of array contraction before applying any transformation. Either way, one of the reasons why the compiler may fail is the presence of array-valued functions that are not inlined, thus inhibiting loop fusion [19]. As Section 3 explains, the HTA library is composed only of headers, and in our codes all the user defined functions were fully specified in header files, thus nothing precludes inlining.

The programming overhead for this optimization on D-dimensional HTAs are two lines for the header and closing curly bracket of the function to apply, D lines for the `for` loops that control the indexing of the D dimensions involved in the operation, and the line with the `hmap` invocation. Sometimes D additional lines are needed to extract the sizes of the loops. In other situations these sizes can be obtained directly from global variables in the loop statements.

The lambda functions approved in the recent C++11 standard, which are already available in many compilers, instead of the traditional functions, permit an even more succinct syntax to express this optimization. It is also important to notice that this optimization is seldom used isolatedly for a single assignment. That is, the function used in an `hmap` invocation often includes several loop nests associated to consecutive operations on the corresponding local tiles of the input HTAs. This allows to reuse the function definition and the sizes extracted for several operations in sequence, thereby diminishing considerably the programming effort required by the optimization.

4.1.2. Avoid the construction and destruction of temporary HTAs

As with regular arrays, the programmer can reduce the memory requirements of an application using only the minimum number of HTAs in the algorithm. In addition, building and destroying them as seldom as possible also helps to increase performance. A typical example of this would be building a temporary HTA used inside each iteration of a loop before a loop, rather than creating and destroying it in each iteration. The optimization in Section 3.2 often reduces this overhead, but the number of HTAs in use could be larger than the pool size, making it ineffective, and even if that is not the case, some time is needed to verify the conformability of the requested HTA with one of the HTAs in the pool. The optimization proposed here requires a single line for building the temporary HTA in an outer scope, as the example in Fig. 9 shows, and it only requires identifying the HTA as a temporary which is repetitively built and destroyed with the same dimensions and structure.

4.1.3. Using the storage of dead HTAs for other purposes

The situation when an HTA goes out of scope can be detected by the library, as described in Section 3.2. However, knowing in advance that the contents of an HTA are not going to be used in the future requires user information. We have noticed that this knowledge is specially useful in the implementation of HTA primitive operations that involve communication such as `transpose` or `permute` (explained in Section 2.5) that require the internal use of intermediate buffers for the communication and copy of data. If the input HTA to these functions is known to be dead, its storage can be used to provide these buffers. Thus, we have extended these functions so that the programmer can provide this information through an additional boolean parameter, which informs whether the input HTA can be used as internal buffer or not. For example, if the contents of HTA `X` are no longer needed after the execution of the code in Fig. 5(b), the permutation operation can be written as `Y = X.dpermute([3,2,1],true)`. This overwrites the contents of `X` with temporary data instead of using additional buffers for the communications, resulting in memory savings and performance improvement. Of course, the programmer can still use the basic implementation of these primitive operations where this information is not provided, forcing to allocate and deallocate the temporary. This technique just involves specifying a boolean flag in the interface of some function, therefore its overhead is 0 SLOC.

4.2. Aggressive inlining and simplification of HTA operations

This optimization is particularly important for the indexing operations. This problem is similar to that of optimizing the computation of the address from a typical array index, with the difference that computing the address from the index space of HTAs is more complicated.

We typically wrote assignment operations using the array notation, as shown in the example in Fig. 3. As explained in Section 4.1.1, computations such as `D = A + B + C` were written using `hmap` and a function that iterates on the elements of each tile performing the corresponding computation in order to avoid the creation of temporaries. This function uses the HTA operator `()` in order to access each individual scalar element of a tile. This is expensive since the operator has to retrieve in each invocation the pointer to the data stored in the HTA and the data about the mapping of the raw data on memory, perform the computation, and return the desired element. An aggressive compiler could be able to inline the code of this

```
for(i=0; i < N; i++) {                    a = hta(/*size like b and c*/);
  a = b + c; /* builds a */  optimize ⟹   for(i=0; i < N; i++) {
  /* use temporary a*/                       a = b + c;
 }                                           /* use temporary a*/
                                            }
```

Fig. 9. Avoidance of creation and destruction overheads by building temporary HTAs in outer scopes.

method in the different invocations in the main loop of the function, perform common subexpression elimination, and move out of the loop the retrieval of the internal structure of the HTA the computations use. Unfortunately none of the compilers we tried were able to do this, so we had to write our functions applying this process manually, as Fig. 10 shows with an optimized version of the function `f` used in Section 4.1.1.

Another solution, used by other libraries that suffer this problem, is to detect common situations (e.g. that all the arrays in the expression have a unit stride in every dimension), and use code optimized for these cases inside the library [13]. This alternative is elegant, but not general, and it can lead to missing several optimizations that the library cannot apply on its own and that the compiler cannot detect because of the code obfuscation involved [14].

This technique is very mechanical, as it consists in applying inlining manually, so the only knowledge needed is the interface for the functions that supply the data involved, which we discuss now. Namely, for each D-dimensional HTA whose indexing is going to be inlined and simplified, this optimization requires $D + 2$ SLOCs in general. One line would get the pointer to the raw data; another one would retrieve the object that holds the information on the mapping of the tile elements on memory, and D lines would be required to store in different variables the stride for each dimension. In practice the stride for the least significant dimension is almost never retrieved because it is known to be 1. Also, while the example in Fig. 10 is a very generic code in which each HTA used may require different strides for its indexing, the most common situation by far is that the HTAs that are processed together have the same sizes, and therefore access strides. This way, the optimization requires $D + 2$ lines (normally $D + 1$, dismissing the least significant dimension) only for the first HTA, while only the line that retrieves the pointer to the raw data is needed for the other ones. Finally, as we said when discussing the programming cost of the elimination of temporaries by using `hmap`, a `hmap` function very often comprises several loops nests in sequence operating on the input tiles. This way, the cost just discussed is only paid for the first loop nest, and the subsequent ones just reuse the information gathered for the first one. Also, although it does not involve additional lines, applying the linearized indexing to a native pointer involves more work than the straightforward indexing of an array. This can be solved in two ways. One is defining a macro to apply the indexing. The other one is defining the native pointer as a pointer to an array with the sizes extracted from the HTA mapping object. The indexing of this pointer is then identical in syntax and therefore complexity to indexing a native array.

4.3. Programming techniques for irregular HTAs

Many algorithms are best expressed with irregular HTAs, that is, HTAs with tiles that have different sizes. Examples of such algorithms are the NAS benchmark IS we study in Section 5.2 or several of the algorithms described in [2]. In these applications an HTA has to be partitioned repetitively in different irregular ways, or an irregular HTA must be built from scratch several times, potentially with a different tiling each time. A solution for these situations, particularly for the latter, is to define the irregular HTAs initially with empty tiles and later, once the irregular partitioning is known, fill them in dynamically allocating the corresponding tiles of different sizes. Fig. 11(a) illustrates a situation of this kind, in which `allocShell` builds an HTA with `n` empty tiles. The tiles are later created so that the `i`-th tile has a variable size depending on the values `v(i)` and `w(i)`. If the process needs to be performed several times, these HTAs must be deallocated before proceeding to allocate the tiles for the next partitioning, which can be a very expensive process.

The HTA library provides two techniques to avoid these overheads. The first one is dynamic partitioning [2], which has been explained in Section 2.5 and illustrated in Section 2.6. The programming effort of this technique can be even a single SLOC to apply `part` or `rmPart` as we have seen.

The second technique is hierarchical indexing, which consists in choosing different indices in different tiles in an already existing HTA. This requires indexing an HTA with an object of a special class called `HTAIndex` which allows to store a different set of indices to select for each tile. For example the instructions in Fig. 11(b), where `A` is an already existing HTA with

```
function f(HTA D, HTA A, HTA B, HTA C) {
  pd = .. // pointer to the beginning of D
  pa =  .. // pointer to the beginning of A
  pb =  .. // pointer to the beginning of B
  pc =  .. // pointer to the beginning of C

  N = size(D)
  sd = stride(D)
  sa = stride(A)
  sb = stride(B)
  sc = stride(C)

  for i=1 to N
    pd[sd * i] = pa[sa * i] + pb[sb * i] + pc[sc * i]
}
```

Fig. 10. Implementation of functions requiring index computation to avoid overheads.

```
B = allocShell([n]);
for i=1 to n
  B{i} = hta({[w(i) - v(i) + 1]});
```

(a)

optimize ⟹

```
hi = HTAIndex(n)
for i=1 to n
  hi(i) = v(i):w(i)
B = A(hi)
```

(b)

Fig. 11. Optimization using hierarchical indexing.

`n` tiles, allow to select a different range of elements `v(i)` to `w(i)` for each one of the `i`= 1,...,`n` tiles of `A` as contents for the corresponding `n` tiles of the resulting irregular HTA `B`. `B` only requires memory to allocate its metadata, since its low level tiles actually correspond to data in `A`. This way this mechanism allows to create irregular sized HTAs at runtime that actually point to data provided by another HTA as long as both HTAs have the same number of tiles in every dimension and the requested sizes are not larger than those of the indexed HTA. This mechanism is more appropriate than dynamic partitioning when what is needed is to choose different ranges of elements from already existing tiles. Also, with this technique the resulting HTA need not (and, in general, will not) keep the same total size as the indexed one.

As seen in the example, building a *D*-dimensional irregular HTA from another HTA by applying hierarchical indexing typically requires $D + 3$ SLOCs. One is the definition of the `HTAIndex` object that stores the hierarchical index. This object can be filled in with *D* `for` loops that iterate on the tiles that are going to be indexed. The body of the innermost loop stores in this object the set of indices to retrieve from each tile, which requires a single line. Finally, the HTA is built indexing the existing HTA with the `HTAIndex` object.

4.4. Synchronization

Sometimes applications provide opportunities to overlap different communications or communications with computation. As we explained in Section 2, in HTA programs communication occurs either inside a method or in an assignment statement of tiles that map to different processors. Our implementation provides the programmer with two kinds of assignments: the regular (represented in this paper with a = operator) and the split-phase assignment (represented here with operator ← [20]). During an HTA assignment `(A = B)`, if `A` and `B` have different processor mappings, the tiles from `B` are sent to the processors that own the corresponding tiles of `A`. The home nodes of the elements of `A` wait (block) until they receive the message from the home nodes of the elements of `B`. However, with an HTA assignment `A` ← `B` the communication is asynchronous and no processor needs to wait for the assignment operation to complete.

By default assignments are of the type =. Assignments of type ← can be specified by the programmer using the call function `async`. A `sync` statement should be explicitly invoked at a later point to indicate that all the assignments in progress must be finished after the sync instruction. If the assignment statements between the `async` and `sync` function calls have aliases, the result will be undefined. An example extracted from the Jacobi stencil computation is shown in Fig. 12, where each of the processors that own tiles `1:n` − `1` will have two communication operations executing concurrently. Thus, the processor that owns tile `i` will communicate with the processor that owns tile `i + 1` and with the processor that owns tile `i` − `1`. As seen in the example, this optimization requires two SLOCs: one for indicating the beginning of the region that allows split-phase assignments, and another one marking the synchronization point where such assignments must have completed.

4.5. Fusion of representation of tiles and management overhead

Each HTA needs metadata about its logical structure, physical layout (addresses) and distribution. This metadata is a memory overhead which additionally needs to be accessed on every HTA access. As a result, algorithms that would benefit from using many small tiles (matrix products, transpositions, etc.) result in performance degradation when those tiles are expressed as HTAs because the added overhead to access the metadata that keeps tile structure cannot be amortized by the small amount of computation performed. Also the larger memory footprint of the program increases the cache miss rates. As a result, we advocate manual tiling for small tile sizes below a certain threshold.

Tiling is a widely known optimization whose semantics, requirements and implementation are well documented (e.g. [21]). This way, the programming cost of manually tiling the *N* dimensions of a computation requires *N* SLOCs for the loops that control the blocking of each dimension. If buffering is going to be used to store a tile of a read-only HTA for the blocked

```
B{1:n}(0) = B{0:n-1}(d);
B{0:n-1}(d+1) = B{1:n}(1);
```

(a)

optimize ⟹

```
HTA::async();
B{1:n}(0) = B{0:n-1}(d);
B{0:n-1}(d+1) = B{1:n}(1);
HTA::sync();
```

(b)

Fig. 12. Optimization using synchronous communication.

computation, in which D of its dimensions lie within the blocked region of the computation, then $D+2$ more SLOCs are needed. One defines the temporary array to store the data in the tile, D manage the loops to perform the copy, and another line makes the copy itself. If the data tiled is read as well as written in the computation, then another $D+1$ lines are needed to perform the copy the data out of the temporary buffer to the HTA.

4.6. Overlapped tiling

The parallelization of stencils requires the exchange of boundary values between different tiles. This requires shadow or overlap regions [22] in each chunk to hold them. Although these regions can be defined and handled manually, the HTA library provides a new language construct, overlapped tiling, to automatically handle them [2,23]. Using the overlapped tiling construct results in faster codes, because the explicit indexing and assignments to perform the updates are not needed. The performance improvement tends to grow with the number of processors involved, as we will see in Section 5. Fig. 13 illustrates the application of overlapped tiling to a stencil. The code in Fig. 13(a) uses HTAs with tiles of `d` +2 elements, where only the positions 1 to `d` contain actual data of the tile. The elements 0 and `d` +1 are ghost regions that store a copy of the last actual element of the preceding tile, and the first actual element of the next tile, respectively. The user has to explicitly update these ghost regions whenever necessary with appropriate assignments, resulting in a code like the one in Fig. 13(a). Care must also be taken to index correctly the tiles when the computation is performed, as we see in the last line. The code in Fig. 13(b) builds HTAs with overlapped tiling of one element in each direction, initially filled in with zeros. These HTAs update automatically their ghost regions without user intervention. Two facts simplify the indexing of these HTAs when performing computations. First, by default, i.e., if no index is applied, only the actual elements of the tile are selected, that is, the tile without the ghost regions. Second, the keyword `All` allows to select the actual regions of each tile, and it supports the arithmetic operators + and − to shift the indexed region, as shown in the figure.

While this optimization is described here because the programmer needs to explicitly use overlapped tiles, its usage also increases the programmer's productivity, since manipulating the shadow regions of stencil computations manually is more cumbersome than using the overlapped tiling construct provided in the HTA library. Typically in a D-dimensional HTA the exchange involves all its dimensions. Thus there are $2D$ assignments (in both directions in the D dimensions). Additionally the 2^D corners of the D-dimensional tile could need to be updated in the corresponding neighbors, resulting in 2^D more assignments. Finally, the exchanges could typically be made in parallel, so the user would have tried to optimize them using the optimization for synchronization, which involves two more lines of code. This is all replaced in overlapped tiling by a line to specify the overlapping when the HTA is created, and another line to request the update of the shadow regions when it is needed.

Support for this optimization is already available in some compilers such as dHPF [24], particularly since the inclusion of the SHADOW directive in HPF2 [25].

4.7. Summary

As seen along this section, the user-level optimizations we have identified remove those overheads introduced by HTAs that could not be removed at library level. Table 2 summarizes these overheads and the techniques that are suitable to avoid them indicated by the numbering of the section where they are described. While all the techniques are very general, those marked with a plus sign in the table have been found to be applicable to all the codes analyzed.

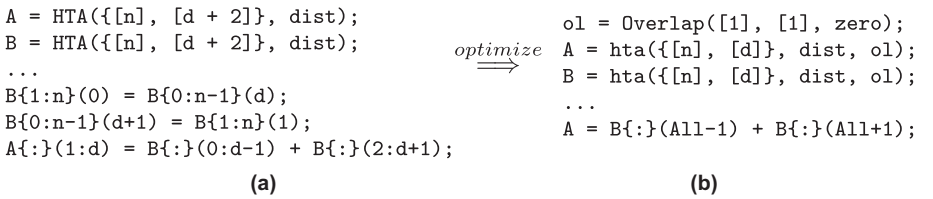


Fig. 13. Overlapped tiling example to implement the stencil `A[i]=(b[i-1]+b[i+1])`.

Table 2
Types of overheads introduced by HTAs and appropriate user-level techniques to avoid them labeled by the section number where they are discussed. Optimizations are marked with a plus sign when they are particularly general.

Overhead	Techniques
Creation and destruction time and memory usage	4.1.1(+), 4.1.2, 4.3, 4.5
Allocation of temporary buffers for communication	4.1.3
Indexing operations	4.2(+), 4.6
Serialization of parallel communications	4.4, 4.6

5. Experimental results

This section evaluates the performance of the HTA library, the impact of the optimizations discussed in the previous sections and the programmability in three stages. In the first one, a microbenchmark is used to illustrate the performance impact of the basic optimizations that apply to all the applications: the avoidance of temporary HTAs (described in Sections 3.1 and 4.1.1) and the simplification of the indexing operations (described in Section 4.2). Second, the NAS benchmarks [3] are used to compare the performance of the original FORTRAN + MPI application with that of the C++ based HTA ones we wrote. Third, a programmability study on these benchmarks is performed.

The characteristics of the machines used to run the experiments are shown in Table 3. The G5 cluster was used for the development and optimization process of the HTA library and benchmarks. After the optimization process, the library was tested in an Itanium and an x86-based cluster. The network interfaces of the x86 cluster suffer high contention when the eight cores of a node are used. For this reason the experiments with this system maximize the number of nodes used. This way, we use one core per node in the configurations for 1–16 processes, and 2, 4 and 8 cores per node for the configurations with 32, 64 and 128 processes, respectively. Table 4 shows the compiler and optimization levels used to compile the NAS benchmarks when using the native FORTRAN + MPI implementation or the HTA library, and the MPI version used in the three cases.

5.1. Performance impact of the common optimizations

Array-based computations produce an output array from the operations performed on one or more input arrays. As a result, the most important optimizations are the avoidance of unnecessary temporaries that will be generated when using a library-based approach (see Sections 3.1 and 4.1.1) and the simplification of indexing operations (see Section 4.2).

To illustrate the importance of these optimizations, we measured the time to execute the expression `A = d * (A + B + C)`, where `d` is a scalar and the other variables are HTAs with 4×4 tiles of 300×300 double precision floating point elements each, in our G5-based machine. The expression required 0.13 s without any expression template implemented. With the expression template implemented in our library (Section 3.1), which saves the temporary only for the last arithmetic operation before the assignment (in this case, the product by the scalar `d`), the time went down to 0.115 s. The approach explained in Section 4.1.1 of encapsulating the whole operation in an `hmap` operation reduced the execution time to 0.05 s. Finally, applying the optimization shown in Fig. 10 and described in Section 4.2, the execution time further reduced to 0.025 s, totaling a speedup of 5.2 with respect to the original naïve implementation. The total speedup achieved by these combined techniques was of similar magnitude in the Itanium and x86 systems, 7.1 and 4.3 respectively.

Due to the high performance impact of these optimizations, we never implemented the NAS benchmarks without them. However, to quantify their impact for a real application, we have implemented an HTA version of the NAS benchmark MG without them and we evaluate it in the next section.

5.2. NAS benchmarks

This section compares the performance of the FORTRAN + MPI NAS benchmarks with that of their HTA-based counterparts and evaluates the performance impact of the optimizations described in Section 4. However, the impact of the optimizations described in Sections 4.1.1 and 4.2 is only evaluated for benchmark MG, because we never implemented HTA versions of the other NAS benchmarks without them due to the high performance impact of these optimizations on array-based codes, as shown in the previous Section. Similarly, all the optimizations described in Section 3 were part of the HTA library from the beginning.

Table 3
Hardware characteristics of the machines used for the experiments.

Proc. Type	Freq.	# of Nodes	# Cores per Node	Memory per Node	Network
G5	2 GHz	128	2[1]	4 GB	Myrinet
Itanium Montvale	1.6 GHz	142 Integrity rx7640	16	128 (max of 8 GB per core)	Infiniband 4x DDR, 20Gbps
x86 Xeon	2.3 GHz	16	8	8 GB	Infiniband 4x DDR, 16 Gbps

[1] Only one core per node was used.

Table 4
Compilers, flags and MPI libraries used for the experiments.

Proc. type	FORTRAN compiler	HTA compiler	MPI library
G5	g77 -O3 (v. 3.3)	g++ -O3 (v. 3.3)	MPICH
Itanium	ifort -O3 (v. 10.1)	icpc -O3 (v. 10.1)	HP MPI
x86	gfortran -O3 (v. 4.3)	g++ -O3 (v. 4.3)	Open MPI

Figs. 14–19 show the running time of the NAS benchmarks EP, FT, CG, IS, MG and LU for problem size C on the machines described in Table 3, using from 1–128 processors. For each benchmark the plots show the execution times of the original FORTRAN + MPI code (labeled as NAS), and the fully optimized HTA version (labeled as HTA). A few runs on 1 or 2 processors are missing because the applications break due to lack of enough memory. For MG we also show the runtime of the version without the common optimizations (HTA nohmap) in the Itanium and the x86 cluster, where it is one order of magnitude slower than the optimized version. The slowdown in the G5 system was similar. IS is a very communication-intensive benchmark that overloads the network of the x86 cluster for both the NAS and the HTA versions when we use 128 processors. The delay happens inside the same MPI library call in both codes, which are exactly the same used in the other platforms.

The performance of the optimized HTA version is in most cases similar to that of the FORTRAN + MPI version in the three machines. The expected performance degradation due to the manipulation of the HTA metadata is usually more noticeable in relative terms as the number of processors increases.

The compiler plays also an important role in the difference in the performance between the FORTRAN and the C++ HTA implementation. This way, while for FT the overhead of the HTA with respect to the FORTRAN + MPI version in the G5-based computer (in which we developed and tuned both the library and the applications) and the x86 cluster is quite small, it goes up to about 50% for every number of processors in the Itanium-based computer. This is in fact the combination of benchmark and machine where the HTA gets further from the original NAS application. We have noticed that when the compilation of both versions of FT in the Itanium system uses gfortran/g++ 4.1.2 (also with O3 optimization level) instead of ifort/icpc, the average overhead drops to just 10%, as Fig. 20 shows. This occurs because there are important optimizations, apparently not

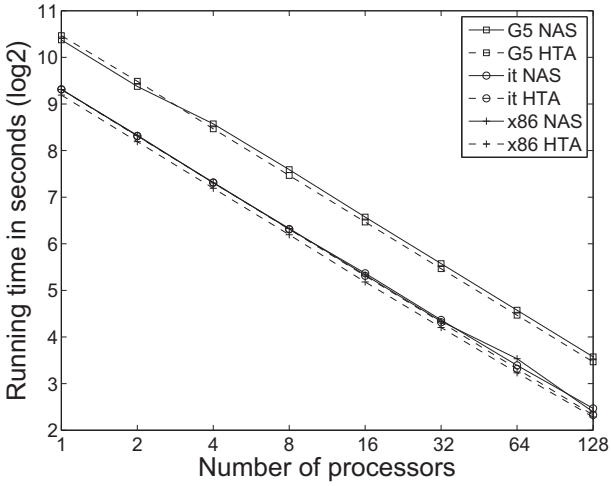


Fig. 14. Performance of EP.

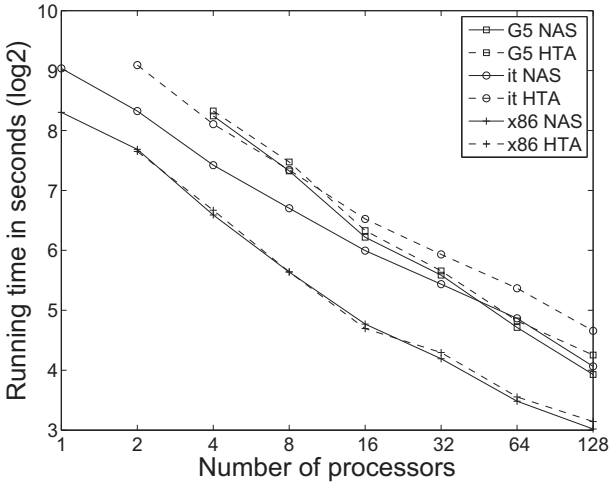


Fig. 15. Performance of FT.

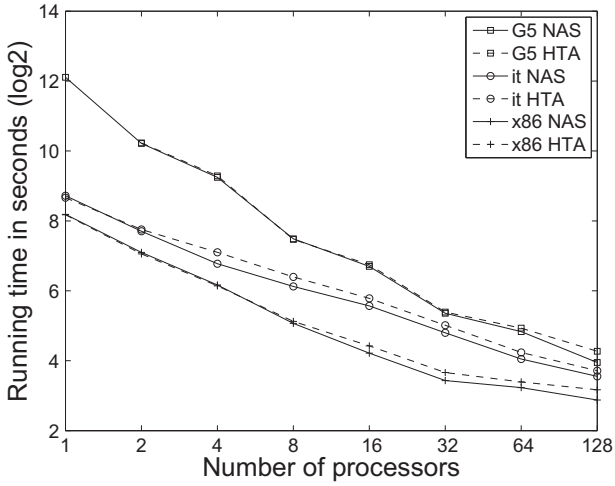


Fig. 16. Performance of CG.

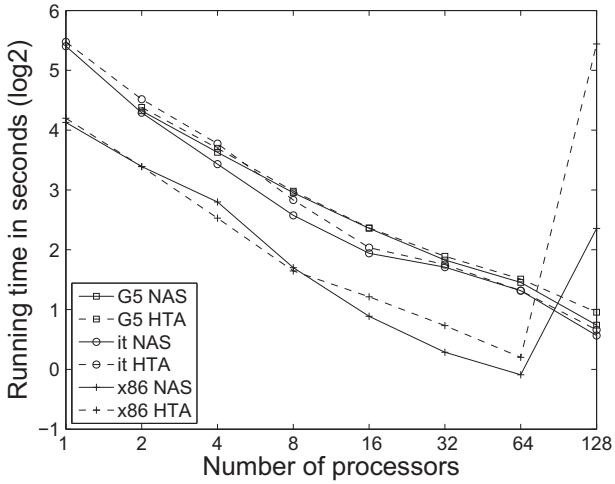


Fig. 17. Performance of IS.

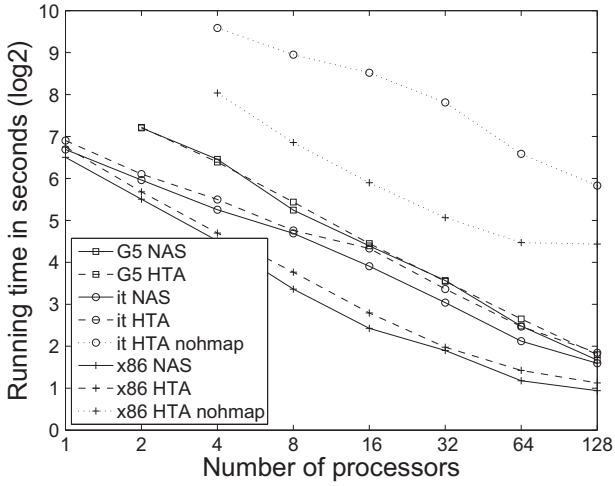


Fig. 18. Performance of MG.

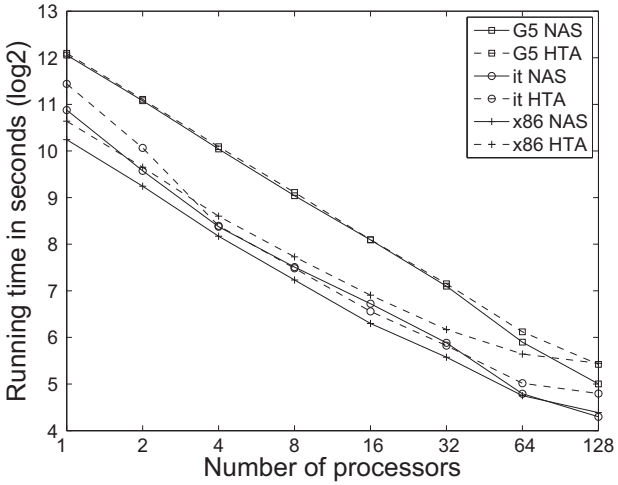


Fig. 19. Performance of LU.

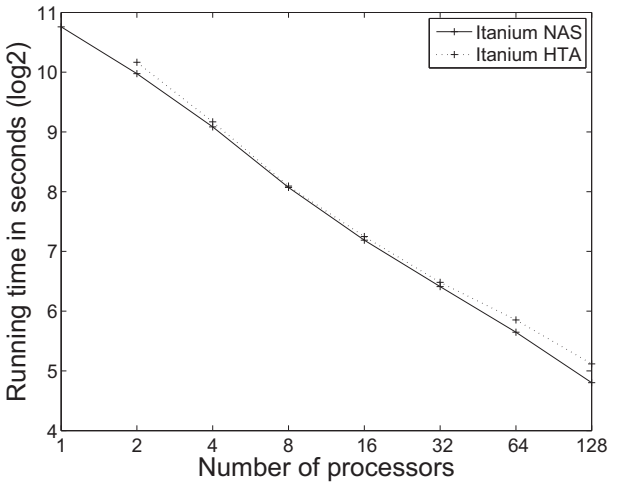


Fig. 20. Performance of FT compiled with gfortran/g++ in the Itanium system

implemented in the GNU compiler, that the FORTRAN code exposes to the ifort compiler, while the C++ code seems to obfuscate the information the icpc compiler needs.

5.2.1. Impact of user-level optimizations

This section analyzes the most important non-common user-level optimizations, i.e., those not covered in Section 5.1, applied in the HTA versions of the NAS benchmarks. Fig. 21 represents the impact of each one of them as the average slowdown the corresponding HTA application would experience if such optimization had not been applied. EP, an embarrassingly parallel program that tabulates pairs of uniformly distributed pseudo-random numbers, only benefits from the most common optimizations evaluated in Section 5.1, so there are no separate optimizations for it.

The FT application solves partial differential equations (PDE) using forward and inverse Fast Fourier Transform on a 3D array. The user can apply two specific optimizations in this code, represented as *FT tiling* and *FT mxr* in Fig. 21. The first one is manual tiling. To compute the 1-D FFTs along each dimension of the array, the FT code uses tiles of 16 by D elements, where D is the size of one of the dimensions of the 3D array in the FORTRAN + MPI version. This way for example, for the class C of the problem considered in our experiments, in which a $512 \times 512 \times 512$ array is used, a total of $512 \times 512/16 = 16384$ identical HTAs (but with different content) are needed. However, preliminary tests declaring these HTAs showed that their storage and management resulted in a crippling overhead. Thus, no tiling was applied for the FFTs in our initial implementation. To achieve a performance more similar to that of the FORTRAN + MPI version for this code we had to implement the tiling manually instead of through HTAs, as discussed in Section 4.5. Our results show that this tiling is very helpful for FT for the

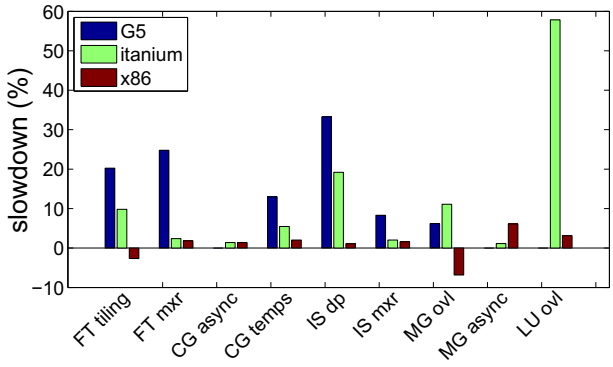


Fig. 21. Slowdown with respect to the optimized version due to the omission of each non-common user-level optimization.

G5 and the Itanium, but it is counterproductive for the x86, probably due to differences in the memory hierarchy and hardware prefetchers. It is possible that a different tile size would have improved the performance of *FT tiling* for the x86 architecture, but to make a fair comparison we have used the same parameters in the HTA as in the original NAS code.

The second user-level optimization was the reuse of an input HTA. This optimization involves the use of the `dpermute` method that allows the user to specify that the input HTA can be used as communication buffer to reduce the memory footprint, as explained in Section 4.1.3. This optimization, which reduces the memory footprint of FT by 20%, is more effective in the G5, where each core has available only 4 GB, than in the other two machines, which provide a minimum of 8 GB/core in the runs with up to 16 processors. In the runs with $N > 16$ processors the Itanium continues to provide 8 GB/core, while in the x86 cluster the memory of each 8 GB node is shared by $N/16$ processors, but at that point the data to fit in each node is also 16 times smaller. If FT is run in this x86 cluster minimizing the number of nodes used, i.e., solving the whole problem with up to 8 processors in the same node, this optimization becomes critical. We have measured that with that configuration the HTA FT version without this optimization does not fit in memory and its frequent use of the disk swap makes its runtime two orders of magnitude longer than that of the optimized version.

CG with the standard optimizations is already very competitive. Still, we found two extra opportunities for further optimization, named *CG async* and *CG temps* in Fig. 21. The first one consists in overlapping the communications in two consecutive statements by applying the policy described in Section 4.4. We do not have data of the impact of this optimization in the G5 system. The global impact of this optimization in this code is small because the statements affected represent a small portion of the runtime, and they are only executed when the number of processors used is an odd power of 2. The second optimization is the reduction of the number of HTA temporaries, and thus the memory footprint, by reusing dead temporary HTAs to hold new temporary values. Again the G5 is more sensitive to memory-related optimizations because it has less memory. In the Itanium system we maximized the number of cores used per node in the runs, while in the x86 we maximized the number of different nodes used. This way the Itanium system is more sensitive than the x86 cluster to memory optimizations, as for runs with $N \leqslant 16$ processors, the N processors are sharing memory bandwidth, while for runs with more than 16 processors, each group of 16 processors is sharing the bandwidth provided by the main memory. In the x86 for runs with 32, 64 and 128 processors, 2, 4, and 8 cores, respectively, access the same main memory.

Benchmark IS sorts a series of integer keys in four steps: local classification in buckets, determination of how many buckets must each processor receive in order to balance work, an all to all exchange of buckets, and a local ranking of the keys received. The rank of a key is the number of keys smaller than it, thus a sorting is implied.

We find this benchmark of particular interest because it is the only one that (a) needs to generate irregular HTAs, i.e., HTAs with tiles with different sizes and (b) these sizes can only be determined at runtime. The first irregular HTA appears in the preparation of the all-to-all exchange. This stage uses an HTA with `NPROC` tiles, each assigned to each processor, which is subtiled in `NPROC` second level tiles so that tile {`i`}{`j`} contains the keys that processor `i` must send to processor `j`. The exchange itself is performed by the HTA function `htranspose`, which sends tile {`i`}{`j`} of its input HTA to tile {`j`}{`i`} of the output HTA. Notice that both the size of the local buckets in each processor as well as the number of buckets that each processor must process are only known at runtime. Thus, our implementation uses a 2 level HTA whose second level tiles are initially empty, and which are filled with the appropriate portions of the HTA where the local buckets are stored once the distribution is calculated. The second irregular HTA is the output of the `htranspose` operation. The third (and last) one is the one in which the rank is computed for each key assigned to the processor.

The first irregular HTA is lightweight because its tiles come from indexing another HTA, which means they only need memory for their metadata, as the data itself are stored by the indexed HTA. The other two irregular HTAs of our initial implementation were fully allocated in the heap and had to be deallocated later. We avoided this by applying the techniques explained in Section 4.3.

First we removed the need to allocate and deallocate dynamically the output HTA of `htranspose` in each usage. Instead we create it at the beginning of the program with enough size to hold the results of the exchange, just as the C + MPI version

does with its buffer. Then when `htranspose` is invoked, instead of allocating new tiles, dynamic partitioning is used to partition the existing tile associated to the running processor into the different output tiles needed. This is optimization *IS dp* in Fig. 21.

The optimization for the third irregular HTA, *IS mxr* in Fig. 21, emulates the same behavior as the C + MPI IS for the corresponding array. The C + MPI version does not use a new array; it just reuses the one where the local keys had been classified in buckets before the global exchange. So we built our third irregular HTA also from the original HTA used for the local classification in buckets, which is regular, by applying hierarchical indexing.

We have already reasoned on why these optimizations that reduce memory usage are more effective in the G5, then the Itanium, and finally the x86 cluster.

The main computation of MG are stencils which involve 27 neighbors. As with CG, the basic common optimizations already bring this benchmark to a performance comparable to that of FORTRAN + MPI. Two user-level optimizations applicable in the code are the usage of asynchronous communications (Section 4.4), labeled *MG async* in Fig. 21, and the use of overlapped tiling to automate the allocation and management of the shadow regions required by the stencils (Section 4.6), labeled as *MG ovl*. We do not have data for the first one in the G5 cluster. While asynchronous communication is beneficial in the x86 cluster, this is not the case for overlapped tiling in this code. The reason is that our overlapped tiling feature automatically updates all the dimensions of the HTAs at once (three in MG), while in our MG implementation with asynchronous communications we synchronize after the set of assignments that update each one of the dimensions, that is, the assignments for each one of the dimensions are enclosed between `HTA::async` and `HTA::sync`, in the style of Fig. 12. The larger number of simultaneous messages in flight generated by overlapped tiling is counterproductive for the network of our x86 cluster, which gets easily congested.

The LU benchmark applies a Symmetric Successive Over Relaxation (SSOR) algorithm, which is a stencil with loop-carried dependences, what involves processor communication when it is parallelized in a distributed memory machine. Thus, this stencil is pipelined [26]. Our implementation uses a wavefront that processes the tiles along the diagonal (or hyperplane for high dimensions) carrying the dependences in the appropriate order. As in any stencil, shadow regions are needed to keep copies of the data from neighboring tiles, and as in MG, in our library this management can be either manual or completely automated using overlapped tiling (Section 4.6). The comparison between the HTA LU version that manages the shadow regions manually and the automated one in the G5 for problem size B yields an speedup that goes from 1% for one processor to 450% for 64, averaging 100%, as shown in [2]; while class C could not be run without overlapped tiling in that system. The slowdown derived from not using overlapped tiling for class C in the Itanium and the x86 clusters, shown in Fig. 21, was 58% and 3%, respectively. In LU the performance benefit of automated overlapped tiling is larger than in MG. A reason is that in wavefront computations the update of the shadow regions is more complex, since the tiles must update different shadow regions and from different neighbors depending on their location. The version of LU that manages manually the shadow regions uses 54 indexing operations to perform this update in each iteration, which is very expensive. The automatic update removes all these operations sending all the required updates at once. We have already discussed in MG why this is a handicap for the subpar network of the x86 cluster.

Overall, the average impact of the positive optimizations (as the programmer would discard those that turn out to be counterproductive) is 10.6%. Since there are large variations among them, the median value, 5.8%, can be more representative. These values are measured on the total execution runtime, rather than on the specific portion where they have been applied.

5.3. Programmability

We finally evaluate the programmability of HTAs versus the traditional MPI-based implementation as well as the programming overhead of the user-level optimizations. The best productivity metric for this evaluation would be the time needed to develop and tune the codes. We did not gather this metric during the development of the HTA versions, and even if we had done it, we have no information on the time needed to develop the optimized MPI applications of the NPB benchmarks, which makes the comparison impossible. This way we have to resort to metrics that can be extracted from the codes. Fig. 22 counts the SLOCs of the FORTRAN + MPI benchmarks, the optimized HTA implementation, and an implementation that only contains the common optimizations evaluated in Section 5.1. Fig. 23 compares the cyclomatic number [27] for these three versions of the algorithm. This value is $P + 1$, where P is the number of decision points or predicates in a program. The smaller it is, the simpler the program is. An HTA version with no optimizations at all is of no practical interest, as its runtime would typically be one order of magnitude longer than that of the original benchmark. Still we can report that the totally unoptimized HTA version of MG we have used in the preceding section has 781 SLOCs versus the 885 of the fully optimized one and the 1715 of the MPI-based code, and a cyclomatic number of 57, compared to the 88 of the optimal HTA version and the 331 of the MPI code.

Fig. 22 shows how overlapped tiling reduced the SLOCs of MG and LU. It increased the cyclomatic number of the LU in Fig. 23 mostly due to the inclusion of conditionals to adjust the processing size of tiles in the corners of the HTA. The other non-common user level optimizations have some programming impact, but it is small, as one would expect from the additional code that we estimated for each one of them in Section 4.

Overall, optimized HTA codes are between 12% (LU) and 54% (CG) shorter than their FORTRAN + MPI counterparts, with an average of 33%. The degree of reduction to expect depends mainly on two factors. The first one is the correspondence of

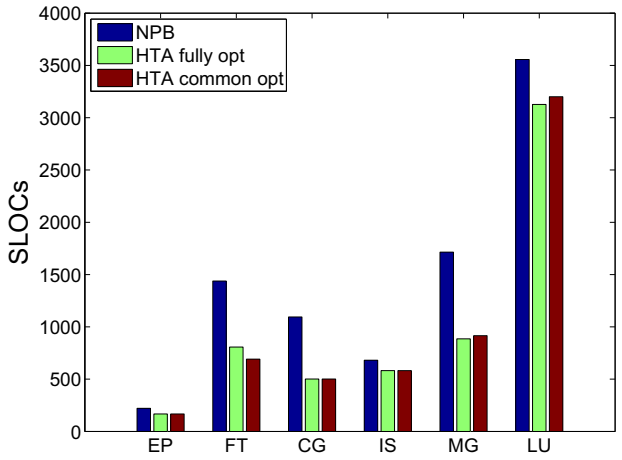


Fig. 22. Comparison in SLOCs between the original NPB benchmarks, the HTA optimized versions and the HTA versions only with the most common optimizations.

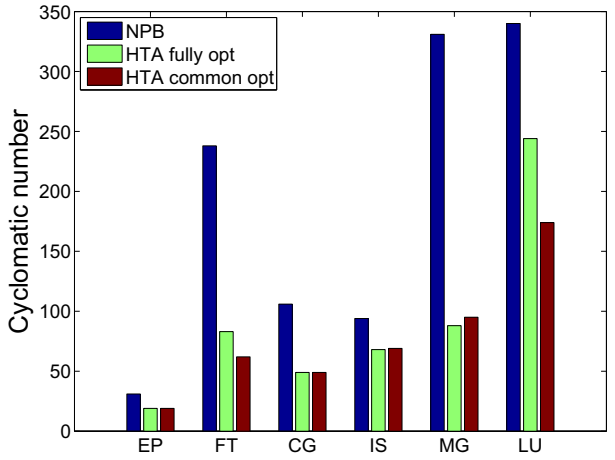


Fig. 23. Cyclomatic number in the original NPB benchmarks, the HTA optimized versions and the HTA versions only with the most common optimizations.

algorithm steps to standard matrix operations. When this is the case, the operations can be typically expressed with a single line with HTAs; otherwise `hmap` invocations and user-defined functions are needed. The second factor are the number and type of communications. HTA operators such as `dpermute` (see Section 2.5) provide functionalities that require dozens or hundreds of lines of FORTRAN + MPI code, while a reduction on a single element across all the processors can be expressed with a single line in both environments.

6. Related work

HTAs are one of many proposals to improve the programmability of parallel applications. Other library-based projects that provide a global view of the data structures are Global Arrays [28] and POET [29], although their SPMD programming style and the requirement of explicit synchronizations complicates their usage. The POOMA library [12] offers a single-threaded view of the program execution, but it does not provide mechanisms to refer to tiles or to decompose them hierarchically in a natural way. From the language side, there has been an emergence of the PGAS approach (Partitioned Global Address Space) [30–32], which offers a global view of data as well as information on the locality of the accesses to those data by the different processors. All these languages follow an SPMD paradigm and do not have tiles and hierarchical decomposition as a first-class concept.

Most research on the implementation and optimization of libraries has been focused on low level libraries such as MPI [9], as they are largely used for parallel programming. Top level libraries [29,12] have received less attention, probably because traditionally their performance has not been competitive with that of other approaches [14], which has led to less widespread usage. Research in this area has given place to active libraries [33]. These are metaprogrammed self-optimizing libraries that play an active role in the compilation process to improve performance by means such as expression

templates [11], programmable syntax macros [34] and meta-object protocols [35]. For example, the MTL [36] and QUAFF [37] have in common with the HTA that they rely on C++ template-based metaprogramming, which enables among other optimizations, the usage of static polymorphism, thus avoiding the costly overhad of dynamic dispatch. As illustration of the importance of this optimization, the C++ skeleton library MUESLI reports in [38] an overhead due to dynamic polymorphism between 20 and 110 percent for simple applications.

As seen along this paper, cross-component optimization is essential to attain reasonable performance while keeping the modularity and encapsulation obtained by writing programs based in the composition of components provided by these libraries. The cross-component optimizations we have found to be of most interest for the HTA library are the loop transformations, particularly emphasizing loop fusion and array contraction [39–41]. Active libraries can rely on advanced techniques such as delayed evaluation [42], runtime code generation [43], and even incorporate liveness analysis [41] or combine at runtime dependence metadata to build polytope representations [40] to support these optimizations. A further step in library-level optimization is the usage of compilers that rely on user provided annotations and a better dataflow analysis than the one a pure active library approach can achieve [44].

Finally, as we have discussed along the paper, some optimizations in the HTA build on techniques used to compile data parallel languages [26]. This way, for languages like HPF and Fortran-D, compilers synthesize message passing communication operations and manage local buffers. Interprocedural analysis can reduce the frequency and volume of communication significantly [45] over a naïve approach. In many situations, a compiler can even identify collective communication patterns that can be implemented very efficiently [46]. In the HTA library, such communication optimization largely results from the way programmers express operations on arrays. If a programmer follows the idiom of using the `hmap` operator, the generic implementation of that operator avoids temporaries, guarantees that buffers are reused and communication is coarsened or optimized using collectives (Section 3). The effectiveness of these optimizations is not in the hands of a complex static program analysis but in the hands of the programmer who conceives the application. Hence the process of optimization is not fully automated but we feel that it is effective and transparent to the programmer. A possible concern is that the programmer may not use the functionality of the HTA library efficiently such that all optimization can become effective at run time.

7. Conclusions

Libraries have several advantages over compile-time approaches as a means to express parallelism, their main drawback being the reduced performance coming from the need to perform all their operations at runtime and the lack of dataflow analysis. In this paper we have described two kinds of techniques to make competitive the HTA library with the fastest implementation available of typical parallel applications such as the NAS benchmarks. First, many techniques were applied at the library level, some being general mechanisms that can benefit other libraries such as templates; others being specific, such as reusing temporary HTAs. Second, there are the HTA specific user-level optimizations. We indicate which are the appropriate ones to address each kind of overhead introduced by HTAs as well as an indication of their programming cost and a measurement of the performance improvement we saw when applying each one of them, therefore providing very useful guidelines to programmers.

After the optimizations, the HTA programs were still 33% shorter in terms of lines of code than their FORTRAN + MPI counterparts and had a 48% smaller cyclomatic number, which measures the number of conditions in a code. As for performance, they were on average only 4.4% slower than the MPI-based codes in the cluster where the optimization process took place, and 18.1% and 14.2% in an Itanium-based supercomputer and an x86-based cluster, respectively, in which the role of the backend compiler and the differences between the ease of analysis of FORTRAN and C++ proved to be much more crucial. This supports the view that an HTA-aware compiler is the way to achieve better performance. Exploring this possibility is our future work.

Acknowledgments

This material is based upon work supported by the Xunta de Galicia under the project INCITE08PXIB105161PR, by the Spanish Ministry of Science and Innovation, cofunded by the FEDER funds of the European Union, under the grant TIN2010-16735, as well as by the National Science Foundation under Awards CNS 1111407, CCF 0702260, and CNS 0720594, and by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign (the center is sponsored by the Intel Corporation). The authors want to acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputer for this paper, as well as Guillermo L. Taboada and Roberto Rey for their help to configure the pluton cluster belonging to the Computer Architecture Group of the Universidade da Coruña for our experiments. Finally we want to thank the anonymous reviewers for their suggestions, which helped improve the paper.

References

[1] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B.B. Fraguela, M.J. Garzarán, D. Padua, C. von Praun, Programming for parallelism and locality with hierarchically tiled arrays, in: Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'06), pp. 48–57.
[2] J. Guo, G. Bikshandi, B.B. Fraguela, M.J. Garzarán, D. Padua, Programming with tiles, in: Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08), pp. 111–122.
[3] N. Aeronautics, S. Administration, NAS Parallel Benchmarks.<http://www.nas.nasa.gov/Software/NPB/> (last accessed 20.05.11)

[4] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B.B. Fraguela, M.J. Garzarán, D. Padua, L. Rauchwerger, Design and use of htalib – a library for hierarchically tiled arrays, in: Proc. 19th Int. Workshop on Languages and Compilers for, Parallel Computing (LCPC'06), pp. 17–32.
[5] G. Bikshandi, Parallel Programming with hierarchically tiled arrays, Ph.D. Thesis, UIUC, 2007.
[6] J. Guo, Exploiting Locality and Parallelism with hierarchically tiled arrays, Ph.D. Thesis, UIUC, 2007.
[7] J.J. Barton, L.R. Nackman, Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples, Addison-Wesley Longman Publishing Co., Inc., 1994.
[8] J. Reinders, Intel Threading Building Blocks, O'Reilly, 2007.
[9] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, second ed., MIT Press, 1999.
[10] J.C. Brodman, G.C. Evans, M. Manguoglu, A. Sameh, M.J. Garzarán, D. Padua, A parallel numerical solver using hierarchically tiled arrays, in: Proc. of the Int. Workshop on Languages and Compilers for, Parallel Computing (LCPC'10).
[11] T.L. Veldhuizen, C++ templates as partial evaluation, in: Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), pp. 13–18.
[12] J.V.W. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjee, W.F. Humphrey, S.R. Karmesin, K. Keahey, M. Srikant, M.D. Tholburn, POOMA: a framework for scientific simulations on parallel architectures, in: Parallel Programming in C++, MIT Press, 1996, pp. 547–588.
[13] T.L. Veldhuizen, Arrays in Blitz++, in: Proc. Second Int. Symp. on Computing in Object-Oriented Parallel, Environments (ISCOPE'98), pp. 223–230.
[14] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W.J. Humphrey, J. Reynders, S. Smith, T. Williams, Array design and expression evaluation in POOMA II, in: Proc. Second Intl. Symp. on Computing in Object-Oriented Parallel, Environments (ISCOPE'98), pp. 231–238.
[15] M.J. Wolfe, Optimizing Supercompilers for Supercomputers, The MIT Press, 1989.
[16] G. Roth, K. Kennedy, Loop fusion in high performance fortran, in: Proc.12th Int. Conf. on Supercomputing (ICS'98), pp. 125–132.
[17] G.R. Gao, R. Olsen, V. Sarkar, R. Thekkath, Collective loop fusion for array contraction, in: Proc. Fifth Int. Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, 1993, pp. 281–295.
[18] E.C. Lewis, C. Lin, L. Snyder, The implementation and evaluation of fusion and contraction in array languages, ACM SIGPLAN Notices 33 (1998) 50–59.
[19] J.D. McCalpin, A case study of some issues in the optimization of Fortran 90 array notation, Scientific Programming 5 (1996) 219–237.
[20] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, Parallel programming in split-c, in: Proceeding of Supercomputing (SC), pp. 262–273.
[21] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.
[22] M. Gerndt, Updating distributed variables in local computations, Concurrency: Practice and Experience 2 (1990) 171–193.
[23] J. Guo, G. Bikshandi, B.B. Fraguela, D.A. Padua, Writing productive stencil codes with overlapped tiling, Concurrency and Computation: Practice and Experience 21 (2009) 25–39.
[24] D.G. Chavarría-Miranda, J.M. Mellor-Crummey, An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications, in: Proc. 2002 Int. Conf. on Parallel Architectures and Compilation Techniques (PACT '02), pp. 7–17.
[25] High Performance Fortran Forum, High Performance Fortran Language Specification, version 2.0, 1997.
[26] S. Hiranandani, K. Kennedy, C.-W. Tseng, compiler optimizations for Fortran D on MIMD distributed-memory machines, in: Proc.of Supercomputing '91, pp. 86–100.
[27] McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (1976) 308–320.
[28] M. Krishnan, B. Palmer, A. Vishnu, S. Krishnamoorthy, J. Daily, D. Chavarria, The Global Arrays User's Manual, 2010.
[29] R.C. Armstrong, A. Cheung, POET (Parallel object-oriented environment and Toolkit) and frameworks for scientific distributed computing, in: Proc.of 30th Hawaii International Conference on System Sciences (HICSS 1997), Maui, Hawai, pp. 54–63.
[30] R.W. Numrich, J. Reid, Co-array Fortran for parallel programming, SIGPLAN Fortran Forum 17 (1998) 1–31.
[31] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, K. Warren, Introduction to UPC and Language Specification, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
[32] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Onward! Track.
[33] T.L. Veldhuizen, D. Gannon, Active libraries: rethinking the roles of compilers and libraries, in: Proc. SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering, Computing (OO'98).
[34] D. Weise, R. Crew, Programmable syntax macros, in: Proc. ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation (PLDI'93), pp. 156–165.
[35] S. Chiba, A Metaobject protocol for C++, in: Proc. Tenth Annual Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'95), pp. 285–299.
[36] J.G. Siek, A. Lumsdaine, The matrix template library: generic components for high-performance scientific computing, Computing in Science and Engineering 1 (1999) 70–78.
[37] J. Falcou, J. Sérot, T. Chateau, J.T. Lapresté, QUAFF: efficient C++ design for parallel skeletons, Parallel Computing 32 (2006) 604–615.
[38] H. Kuchen, A skeleton library, in: Proc. Eigth Int. Euro-Par Conference on Parallel Processing (EuroPar'02), LNCS, vol. 2400, pp. 620–629.
[39] T.J. Ashby, A.D. Kennedy, M.F.P. O'Boyle, Cross component optimization in a high level category-based language, in: Proc. 10th International Euro-Par Conference on Parallel Processing (EuroPar'04), LNCS, vol. 3149, pp. 654–661.
[40] J.L.T. Cornwall, P.H.J. Kelly, P. Parsonage, B. Nicoletti, Explicit dependence metadata in an active visual effects library, in: Proc. 20th Int. Workshop on Languages and Compilers for, Parallel Computing (LCPC'07), pp. 172–186.
[41] F.P. Russell, M.R. Mellor, P.H. Kelly, O. Beckmann, An active linear algebra library using delayed evaluation and runtime code generation, in: Proc. Library-Centric Software Design (LCSD'06), pp. 5–13.
[42] P. Liniker, O. Beckmann, P.H.J. Kelly, Delayed evaluation, self-optimizing software components as a programming model, in: Proc. Eighth Int. Euro-Par Conference on Parallel Processing (EuroPar'02), LNCS, vol. 2400, pp. 666–674.
[43] O. Beckmann, A. Houghton, M.R. Mellor, P.H.J. Kelly, Runtime code generation in C++ as a foundation for domain-specific optimisation, in: Int. Seminar on Domain-Specific Program Generation, LNCS, vol. 3016, pp. 291–306.
[44] S.Z. Guyer, C. Lin, Broadway: a compiler for exploiting the domain-specific semantics of software libraries, Proceedings of the IEEE 93 (2005) 342–357.
[45] G. Agrawal, J. Saltz, Interprocedural data flow based optimizations for distributed memory compilation, Software Practice and Experience 27 (1997) 519–548.
[46] M. Kandemir, P. Banerjee, A. Choudhary, R. Ramanujam, N. Shenoy, A generalized framework for global communication optimization, in: Proc. 12th. Int. Parallel Processing Symp. (IPPS '98), pp. 69–73.