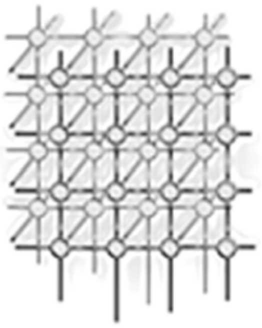
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2008)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1340



Writing productive stencil codes with overlapped tiling[‡]

Jia Guo[1], Ganesh Bikshandi[2], Basilio B. Fraguela[3,*,†] and David Padua[1]

[1]*University of Illinois at Urbana-Champaign, Urbana, IL, U.S.A.*
[2]*IBM, India*
[3]*Universidade da Coruña, Spain*

SUMMARY

Stencil computations constitute the kernel of many scientific applications. Tiling is often used to improve the performance of stencil codes for data locality and parallelism. However, tiled stencil codes typically require shadow regions, whose management becomes a burden to programmers. In fact, it is often the case that the code required to manage these regions, and in particular their updates, is much longer than the computational kernel of the stencil. As a result, shadow regions usually impact programmers' productivity negatively. In this paper, we describe *overlapped tiling*, a construct that supports shadow regions in a convenient, flexible and efficient manner in the context of the hierarchically tiled array (HTA) data type. The HTA is a class designed to express algorithms with a high degree of parallelism and/or locality as naturally as possible in terms of tiles. We discuss the syntax and implementation of overlapped HTAs as well as our experience in rewriting parallel and sequential codes using them. The results have been satisfactory in terms of both productivity and performance. For example, overlapped HTAs reduced the number of communication statements in non-trivial codes by 78% on average while speeding them up. We also examine different implementation options and compare overlapped HTAs with previous approaches. Copyright © 2008 John Wiley & Sons, Ltd.

KEY WORDS: productivity; shadow regions; tiles; overlapped tiling; stencil computations

*Correspondence to: Basilio B. Fraguela, Facultade de Informática, Campus de Elviña, s/n. 15071 A Coruña, Spain.
†E-mail: basilio@udc.es
‡Portions of this paper were previously published in 'Programming with Tiles' by Guo J, Bikshandi G, Fraguela BB, Garzarán MJ, Padua D. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2008.

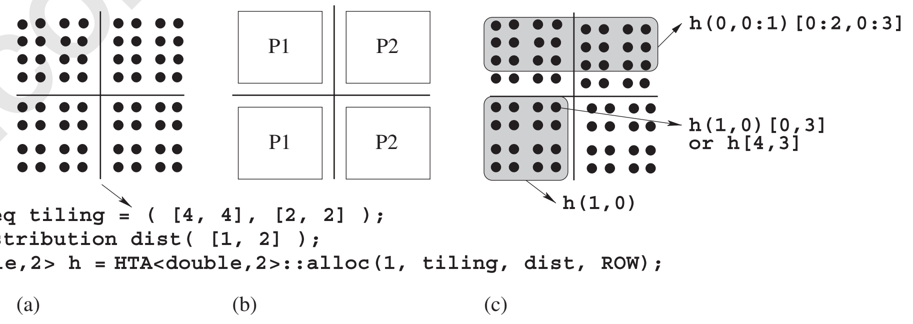
Contract/grant sponsor: National Science Foundation; contract/grant numbers: Awards CCF 0702260, CNS 0509432
Contract/grant sponsor: Ministry of Education and Science of Spain
Contract/grant sponsor: FEDER Funds of the European Union; contract/grant numbers: TIN2004-07797-C02-02, TIN2007-67537-C03-02

1. INTRODUCTION

Stencil computations arise in many scientific and engineering codes and thus have been widely studied. Tiling [1] is often used to improve the locality of these codes and to simplify the coding of communication in their parallel versions. As stencils require a number of neighboring values to compute each result, the computations associated with each tile require data from adjacent tiles. In distributed-memory computers, this leads to the creation of ghost or shadow regions that surround the tile assigned to each processor with copies of data from neighboring tiles. Stencil computations are executed repetitively within a time-step loop, updating the input array for the next iteration with the values of the input array for the current iteration. As a result, shadow regions facilitate the implementation of the tiled stencil computations by simplifying the computation of boundary elements of each tile. Tiled versions are important for parallel execution in distributed- and shared-memory machines and even in sequential computations to exploit data locality. The responsibility for the creation and management of these regions is shared in different ways between the user, the compiler and libraries, depending on the programming environment. Their management can be cumbersome, as shadow regions must be updated to reflect the changes carried out on the values they replicate.

The hierarchically tiled array (HTA) [2] is a data type designed to improve the programmers' productivity by providing a natural representation for sequential and parallel tiled computations. As a part of our ongoing effort to improve the HTAs, we proposed a number of new constructs in [3]. One of them was *overlapped tiling*, which supports shadow regions in the stencil codes, freeing the programmers from most of the details related to the definition and update of such regions. An experimental evaluation, in terms of both performance and productivity, was presented in [4]. In this paper, we expand the discussion of overlapped tiling presented in [4] with a description of its implementation in the HTA library and a comparison with other approaches in detail. We also extend the evaluation with an analysis of the performance implications of the policy chosen to update the shadow regions. A first experience on the learning curve and the impact on the development time of this construct are also reported.

Codes written using our overlapped tiling operations are simpler because the shadow regions are transparent to the programmers. The implementation is also simple, as no data-dependence analysis is needed. Our experiments with non-trivial benchmarks show that using our methods



```
Tuple::Seq tiling = ( [4, 4], [2, 2] );
CyclicDistribution dist( [1, 2] );
HTA<double,2> h = HTA<double,2>::alloc(1, tiling, dist, ROW);
```

Figure 1. HTA allocation, distribution and access.

for overlapped tiling greatly improves readability without negatively affecting the performance. Thus, the HTA versions of the NAS [5] MG and LU benchmarks written with overlapped tiling have 78% fewer communication statements than the original versions. The performance was also improved, as the version with overlapped tiling was faster than the original HTA code for both benchmarks. The average speedup was 6.6% for MG and 103% for LU and was 33% if we ignore one particular case in which the performance of the version without overlapped tiling falls considerably.

The remainder of this paper is organized as follows. Section 2 presents an overview of our HTA data type and its runtime system. Section 3 shows the design and implementation of our *overlapped tiling* construct. Section 4 demonstrates experimentally the benefits of overlapped tiling. Related work is discussed in Section 5. We conclude in Section 6 with a summary.

2. HIERARCHICALLY TILED ARRAYS

HTA [2] is a data type that facilitates programming for parallelism and locality. An HTA is an array partitioned into tiles. These tiles can be either conventional arrays or lower-level HTAs. Tiles can be distributed across processors in a distributed-memory machine or be stored in a single machine according to a user-specified layout. We have implemented HTAs in MATLAB [2] and in C++ [3].

2.1. Construction of HTAs

Figure 1(a) shows the creation of a C++ HTA, with the `alloc` constructor being invoked in the third line of the code. Its first argument specifies that the HTA will have a single level of tiling. The second argument specifies the tiling as a sequence of two tuples indicating, respectively, the shape of the innermost (or leaf) tile and the number of tiles at each level from the lowest to the uppermost. Thus, in our case the HTA has 2×2 tiles of 4×4 elements each. The third argument `dist`, which is optional, specifies how to distribute the tiles across processors, as depicted in Figure 1(b). The last parameter specifies the layout of the HTA. The HTA runtime system allows row major, column major and a *tile* data layout, which places the elements inside a tile contiguously in memory. The data type and the dimension of HTA are template parameters of the HTA class.

2.2. Accessing HTAs

HTAs' indexing is 0-based in C++. In order to simplify the notation, indexing tuples in this paper use the `low:high:step` notation to represent a range, where `step` can be omitted if it is 1. In addition, a single colon can be used in any index to refer to all the possible values for that index.

We overload operator `()` to access tiles and operator `[]` to index elements in an HTA disregarding the tiling structure. Figure 1(c) shows examples of how to access HTA components. The expression `h(1,0)` refers to the lower left tile. The scalar in the fifth row and fourth column can be referenced as `h[4,3]` just as if `h` were not tiled. This element can also be accessed by selecting the tile that contains it and its relative position within this tile: `h(1,0)[0,3]`. In any indexing, a range of components may be chosen in each dimension using the triplets notation. For example, `h(0,1)[0:2,0:3]` selects the first three rows in tile (0,1).

2.3. Assignments and binary operations

We generalize the notion of conformability of Fortran 90. When two HTAs are used in an expression, they must be conformable, i.e. they must have the same topology and the corresponding tiles in the topology must have sizes that allow one to operate them. The operation is executed tile by tile, and the output HTA has the same topology as the operands.

In addition, an HTA is always conformable to a scalar. The scalar operates with each scalar component of the HTA. An HTA is also conformable with an untiled array if each HTA leaf tile is conformable with the array. The untiled array will be operated with each leaf tile.

Assignments to HTAs follow rules similar to those 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, if both are conformable. When the tiles of an HTA are distributed across multiple processors of a distributed-memory machine, assignments involving tiles located in different processors execute communication operations.

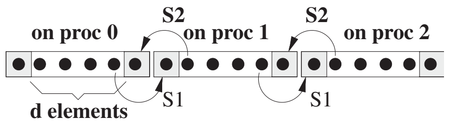
2.4. A simple Jacobi example using HTAs

Figure 2(a) illustrates a 1D Jacobi computation using two HTAs, `A` and `B`. Each of these HTAs contain `n` tiles distributed on `n` processors. Each tile holds `d`+2 values, elements at indexes 0 and `d`+1 in each tile being shadow regions that must be updated before the computation. This update is done in lines 5 and 6. As the source and destination data are on different processors, these assignments imply communications, as Figure 2(b) shows.

This example shows that without further support, the programmer is responsible for explicitly allocating and updating the shadow regions. Furthermore, it is easy to make mistakes while writing the complex indices associated with the update of the shadow regions. This problem grows with the number of dimensions of the arrays.

```
1 Tuple::seq tiling = ( [d + 2], [n] );
2 A = HTA<double,1>::alloc(1, tiling, dist, ROW);
3 B = HTA<double,1>::alloc(1, tiling, dist, ROW);
4 while (!converged) {
5   B(1:n)[0] = B(0:n-1)[d];
6   B(0:n-1)[d+1] = B(1:n)[1];
7   A(:)[1:d] = 0.5 * (B(:)[2:d+1] + B(:)[0:d-1]);
8   ...
9 }
```

(a)



(b)

Figure 2. 1D Jacobi in HTA: (a) HTA code with one level of tiling and (b) shadow region exchanges.

3. OVERLAPPED TILING

Our C++ implementation of HTAs includes a construct, called *overlapped tiling*, which allows the users to specify that the tiles in an HTA overlap to a given extent and even that they want the corresponding shadow regions to be kept updated automatically.

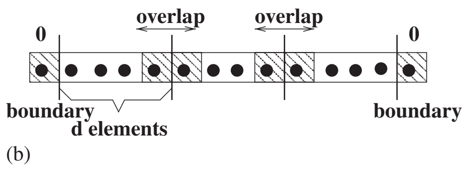
3.1. Syntax and semantics

The HTA constructor, shown in the third line in Figure 1, allows a fifth optional argument that is an object of the class `Overlap`. This object conveys the information about the type of overlapping the user wants in the HTA. Its constructor is

```
Overlap<DIM>( Tuple negativeDir, Tuple positiveDir, boundaryMode mode, bool autoUpdate = true);
```

where `DIM` is the number of dimensions of the HTA to be created. The `negativeDir` specifies the amount of overlap for each tile in the *negative direction* (decreasing index value) in each dimension. The `positiveDir` specifies the amount of overlap for each tile in the *positive direction* (increasing index values). These parameters define the size of the overlap and the size of the boundary region to be built around the array. The third argument specifies the nature of this boundary along every dimension. A value `zero` indicates that the boundary region will be filled with constants. A value `periodic` indicates that the first and last elements along each dimension are adjacent. Finally, the fourth (optional) argument specifies whether the library must automatically update the shadow regions. Its default value is true. Figure 3(a) is the 1D Jacobi in Figure 2(a) using overlapped tiling. This time, the tiles contain `d` elements, but an overlap of one element in each direction is requested. The resulting HTA is depicted in Figure 3(b). The overlapping extends the indexing within each tile according to its length in order to read the elements from the neighboring tiles. The range of indexing for the tiles in the HTA in our example goes from −1 to `d`, both included, as shown in Figure 3(c). The symbol `ALL` may be used to refer to the elements of the tile (range 0:`d`−1 in our example), which we call the *owned* region. The symbol also allows the arithmetic operators + and − to shift the indexed area as shown in Figure 3(c).

```
Tuple::seq tiling = ( [d], [n] );
Overlap<1>   ol( [1], [1], zero);
A = HTA<double,1>::alloc(1, tiling, dist, ROW, ol);
B = HTA<double,1>::alloc(1, tiling, dist, ROW, ol);
while (!converged) {
 A = 0.5 * (B(:)[All-1] + B(:)[All+1]);
 ... ...
}
```
(a)



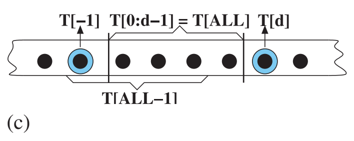


Figure 3. HTA 1D Jacobi using overlapped tiling: (a) the code; (b) the pictorial view; and (c) the overlapped HTA indexing.

Notice that conformability rules only apply to the tile itself without including the shadow regions. Also tiles can only write to their own data and cannot write to the data that belong to the neighboring tiles.

3.2. Shadow region consistency

HTAs allow the programmers to choose either to keep the shadow regions consistent with the data they replicate manually or to rely on the class to do it for them automatically. HTAs support manual updates by means of three methods: `A.update()` performs a synchronized update on HTA `A`, whereas `A.asyncUpdate()` and `A.sync()` perform an asynchronous update.

3.2.1. Update policy

Two strategies can be followed to update the shadow regions when the data they replicate are modified. In [3] we suggested using *update on write*, which performs the update immediately. The other alternative is *update on read*, which performs the update in a lazy fashion, i.e. only when the affected shadow regions are read. Although *update on write* has the advantage of sending the modified data as early as possible, it does not know which neighbor(s) will actually need the data later. Therefore, the modified array elements are sent to every neighbor whose shadow regions overlap with the modified area. On the contrary, *update on read* only updates the neighboring tiles that are needed, although bookkeeping for the status of each tile is required, and the update is delayed until the next read of the neighboring tiles.

We selected the *update on read* policy for two reasons. First, it leads to a minimal number of communications, as only the tiles needed will be updated. This is important for wavefront computations as we will see in Section 4.2.2. Second, the bookkeeping mechanism of this policy is the same as that of the manual update. The automatic *update on read* policy can be accelerated by the user by placing a manual update for the neighboring tiles immediately after a write.

3.2.2. Implementation in the HTA library

The implementation of our *update on read* policy relies on a small table that records for each tile the state of its shadow regions, which we call *shadow-out* regions, and the portions of the tile that neighboring tiles can access through their shadow regions, which we call *shadow-in* regions. We track the status of the shadow regions from both the sender side (shadow-in region) and the receiver side (shadow-out region). This dual tracking is needed because our library uses a two-sided communication layer. If one-sided communication were used, the library would only need to keep track of the receiver side (shadow-out region). The number of entries in the table is equal to the number of neighbors each tile overlaps with. Each entry consists of two fields, shadow-in region status and shadow-out region status. Their values change after read and write operations as shown in Table I. When there is a need for an update (i.e. there is a read operation), the whole shadow-in region is sent to its corresponding neighbor. Notice that as the HTA class provides a global view and a single logical thread of execution, each tile knows every operation performed on it and its neighbors. Therefore, bookkeeping requires no communications.

Table I. Actions for each shadow-in region and shadow-out region on read and write.

Operations	Owner shadow-in region status	Corresponding neighbor shadow-out region status
Owner writes in shadow-in region	Set status to *inconsistent*	Set status to *inconsistent*
Neighbor reads from shadow-out region	If *inconsistent*, send the update, set status to *consistent*	If *inconsistent*, receive the update, set status to *consistent*.

We placed the update actions shown in the last row of Table I in all the HTA functions or overloaded operators accepting an overlapped HTA as an input. Such operators include $+$, $-$, $=$, as well as functions `hmap` and `mapReduce` [3]. The former applies a user-defined function in parallel with the corresponding tiles of its input HTAs, whereas the latter applies on an HTA a *map-reduce* [6] operator, which defines a *map* followed by a reduction. Those functions will update their input HTAs before executing. In addition, functions or operators that write to HTAs invoke the bookkeeping actions listed in the first row of Table I. For example, operator$=$ and `hmap()` add the bookkeeping functions at the end. By inserting update functions before the reads and placing bookkeeping functions after the writes, our library provides automatically updated shadow regions.

Fully integrating overlapped tiling in the HTA required additional changes. For example, when an HTA is indexed, the resulting HTA actually points to the same data of the original one, providing only a new mapping over it. As the HTAs generated by indexing overlapped HTAs are normal HTAs, writing to the shadow regions of the data through these HTAs must be prevented. In addition, the legal reads and writes to the data through these HTAs must trigger the bookkeeping mechanisms described above. All of this is achieved by linking these HTAs to the original overlapped HTA they are derived from, which allows them to use the same data structures to keep the shadow regions consistent. The resulting mechanism is very powerful. For example, even if the neighboring tiles are not a part of the indexed HTA, the library can perform the update or bookkeeping successfully, thanks to the link to the overlapped HTA.

4. EVALUATION

In this section, we evaluate the impact of our approach both in terms of performance and programmer productivity.

4.1. A sequential 3D Jacobi computation

We examined the performance of a sequential 3D Jacobi code for overlapped tiling on a 3.0 GHz Pentium 4 with 8 kB L1, 1 MB L2 and 1 GB RAM. Figure 4 compares the Jacobi code using (1) an untiled 3D array Jacobi implementation in C++ (Untiled), (2) a tiled 3D array Jacobi implementation in C++ (Tiled), (3) an HTA implementation without overlapped tiling (Orig-HTA) and (4) an HTA version using the overlapped tiling construct (OL-HTA). We searched for the best tiling sizes for the Tiled and OL-HTA versions and present the best performance we found. Orig-HTA uses the same tile sizes as OL-HTA so that the only difference between them is the management of the

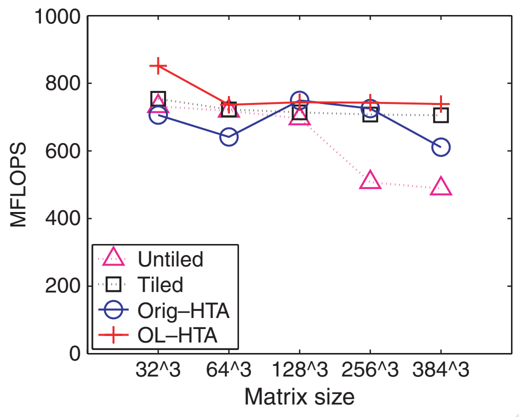


Figure 4. Performance of sequential Jacobi code on an Intel Pentium 4.

shadow regions. The performance of the Untiled version drops when the matrix size increases to 256^3 due to poor data locality. The performance of Orig-HTA suffers because HTAs without overlapped tiling require explicit shadow regions that need to be updated manually, which sometimes offsets the advantages of tiling. The other tiled versions do not experience performance degradation, thanks to tiling and the implicit management of the shadow regions. Compared with the Tiled version, the OL-HTA is slightly faster about 5.74% on average. This is because the Tiled version uses three outer loops to iterate 3D tiles, whereas the HTA class uses a single loop to traverse the tiles.

4.2. Parallel benchmarks: MG and LU

We also applied overlapped tiling to parallel stencil computations. We rewrote the NAS [5] MG and LU benchmarks using our HTA language extension with overlapped tiling. We tested the performance of our implementation as well as the NAS benchmarks written in Fortran and MPI on a cluster consisting of 128 nodes, each with two 2 GHz G5 processors and 4 GB of RAM, interconnected by a high-bandwidth, low-latency Myrinet network. We used one processor per node in our experiments. The NAS codes were compiled with g77, whereas the HTA codes were compiled with g++. −O3 optimization level was used in both the cases.

4.2.1. MG

The MG benchmark is an example of loosely synchronous computation in which all the processors alternate between local computation and synchronous global communications [7]. It uses a finite difference scheme in a multi-grid algorithm to solve the 3D Poisson's equation with periodic boundaries. The main steps of the algorithm are two inter-grid operations: *projection* and *interpolation* and two intra-grid operations: *inversion* and *residual* computation. After each operation mentioned above, communications occur to update the shadow regions.

As we will see in Section 4.3, the MPI MG benchmark from NAS contains more lines of source code for communications than for computations. In the original HTA program without overlapped

```
//NX, NY, NZ are the number of tiles in dimension x,y,z;
//nx, ny, nz are the tile sizes in dimension x,y,z
if (NX > 0)
   u(0:NX-1, 0:NY, 0:NZ)[nx, 1:ny-1,1:nz-1] = u(1:NX, 0:NY, 0:NZ)[1, 1:ny-1, 1:nz-1];
u(NX, 0:NY, 0:NZ)[nx, 1:ny-1, 1:nz-1] = u(0, 0:NY, 0:NZ)[1, 1:ny-1, 1:nz-1];
if (NX > 0)
   u(1:NX, 0:NY, 0:NZ)[0, 1:ny-1, 1:nz-1] = u(0:NX-1, 0:NY, 0: NZ)[nx-1, 1:ny-1, 1:nz-1];
u(0, 0:NY, 0:NZ)[0, 1:ny-1, 1:nz-1] = u(NX, 0:NY, 0:NZ)[nx-1, 1:ny-1, 1:nz-1];
```

Figure 5. Explicit shadow region exchange in dimension x in the original HTA MG program.

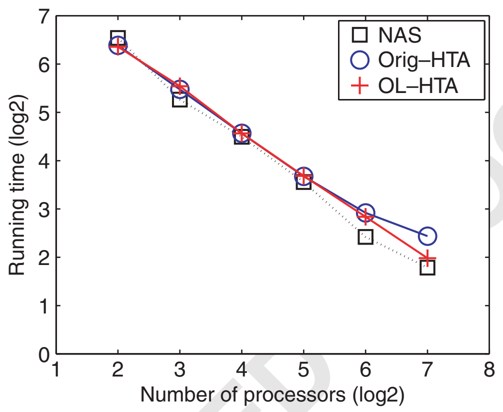


Figure 6. The performance of MG (class C) on a cluster of 2 GHz G5 processors.

tiling, the programmer explicitly managed the shadow regions and periodic boundaries. Figure 5 shows the shadow region exchange for dimension x. In order to perform a correct update for a single dimension, the programmer has to write 48 index expressions. With the overlapped tiling construct, the shadow regions are completely absent from the program.

Figure 6 shows the performances of the three versions of the MG benchmark in NAS class C: the MPI NAS parallel benchmark (NAS), the original HTA program without overlapped tiling (Orig-HTA) and the HTA with overlapped tiling (OL-HTA). It shows that the new language construct does not add overhead to the HTA version, but in fact it is even faster as the number of processors grows. The reason is that it avoids the indexing operations shown in Figure 5 and the corresponding creation and destruction of temporary HTAs. The performance of overlapped tiling is also close to that of the NAS MPI benchmark.

4.2.2. LU

The Symmetric Successive Over Relaxation algorithm used by LU represents a different class of stencil computation, which contains loop-carried cross-processor data dependencies that serialize computations over distributed array dimensions. Such computations are called *pipelined* [8]. We use a wavefront algorithm [9] to compute the tiles the appropriate order.

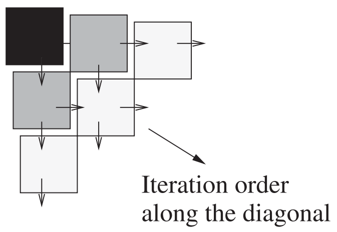


Figure 7. Illustration of diagonal iteration over tiles in the wavefront algorithm.

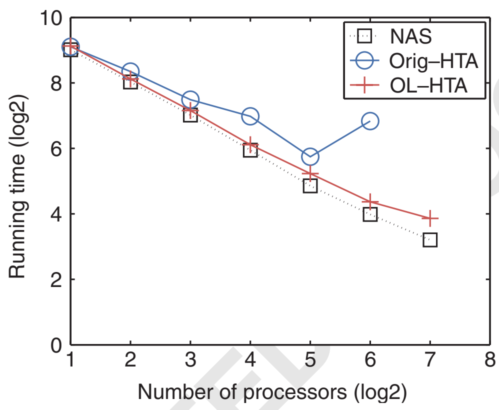


Figure 8. The performance of LU (class B) on a cluster of 2 GHz G5 processors.

Figure 7 shows the iteration order for the 2D case. In each iteration, the wavefront algorithm selects the tiles in the same color and computes them in parallel. After the computation, the shadow regions of the next band of tiles are updated from their north and/or west neighbors. Implementing these updates explicitly is difficult because some tiles need updates from both neighbors, whereas others need them from only their north or west neighbor. In our initial HTA implementation of LU, a set of 54 indexing operations are used to complete the shadow region update in one iteration. With overlapped tiling these operations are unnecessary as the update is automatic. By utilizing the parent field, the bookkeeping mechanism and disabling the automatic update for temporary HTAs, our HTA class can efficiently update the shadow regions with a minimal number of communications.

The performance of LU in class B is presented in Figure 8. The three versions are the same as those of the MG benchmark. The overlapped HTA program again outperforms the original HTA program as it eliminates indexing operations. Its performance is similar to that of the NAS version.

We also used LU to evaluate the performance improvement of *update on read* (UoR) compared with *update on write* (UoW), and the extra performance attainable by overlapping communication and computation by updating manually the HTAs using the asynchronous update functions `asyncUpdate` and `sync` (Async). We chose LU to make this comparison for two reasons. First, not all neighbors need to be updated in the wavefront, so that we can better see the difference

Table II. Execution time of LU for different problem sizes and shadow regions' update policy in a cluster of 2 GHz G5 processors (in seconds).

	Class A			Class B			Class C		
# Procs	UoR	UoW	Async	UoR	UoW	Async	UoR	UoW	Async
2	129.9	130.6	128.7	534.2	536.1	527.9	2214.3	2225.3	2224.9
4	66.5	67.3	65.3	287.5	285.1	281.9	1214.1	1223.3	1207.4
8	30.9	32.0	30.2	145.4	146.4	144.3	593.7	607.1	598.4
16	16.1	17.0	15.9	69.1	71.7	68.3	287.0	294.6	288.5
32	9.9	10.7	9.6	35.8	36.8	34.9	163.3	163.6	159.0
64	7.1	7.7	6.8	21.0	21.7	20.2	79.5	78.8	78.0
128	6.2	7.0	5.8	15.7	16.3	15.1	51.7	50.0	49.2

between UoR and UoW. Second, it provides good opportunities to overlap communication and computation.

The execution times of the benchmark for the classes A, B and C using the three update policies are shown in Table II. As expected, the larger the problem, the smaller the impact of both optimizations, as the weight of communication in the execution time decreases. The total execution time of LU was on average 5.66, 1.99 and 0.32% shorter with UoR than with UoW for classes A, B and C, respectively. Similarly, manual asynchronous updates reduced the total execution time on average by 3, 2.19 and 1.22% with respect to automatic UoR for classes A, B and C, respectively.

The difference between the two approaches grows with the number of processors. For example, for class B, the advantage of UoR over UoW grew from 0.35% with 2 processors to 4.09% with 128, and asynchronous update improvement went from 1.2% with 2 processors to 3.85% with 128. The exception was class C, in which the advantage of UoR or UoW grew up to 2.63% with 16 processors and then declined, being 3.25% slower with 128. We found that although the update operation was faster in UoR than in UoW, the computational part of LU was slower. We think that it is a problem of locality, although we have not been able to measure it. LU calculates some temporaries that are then operated with the HTA object of the update. These temporaries fit in the L1 when 128 processors are used, but into UoR the update happens between the calculation of the temporaries and their operation with the HTA, which displaces some of them from the caches due to the usage of communication buffers. UoW, on the other hand, updates the HTA after the temporaries have been used, thus achieving better locality.

These figures seem to suggest that the update policy has a relatively small impact on performance. Actually, this impact depends to a large extent on communication times, the relative speed of communication and computation, and, to a minor extent, on the MPI implementation.

In order to prove this, we conducted the same experiments on LU in a cluster with gigabit ethernet, which is quite slower than the Myrinet used above. This cluster consists of 8 nodes with two dual core 3.2 GHz Pentium Xeon 5060 processors, faster in general than the 2 GHz G5 processors used before. We ran experiments using 2–32 processes. When more than 8 processes were used, 2 or 4 of them shared the memory; thus, the environment was hybrid. We do not include here the measurements due to space limitations. The performance improvement ratios for UoR over UoW grew to 16.27, 7.5 and 4.21% for classes A, B and C, respectively, as compared with those obtained in the other cluster. This is not surprising given that this network is much slower with respect to its processors. The improvement always grew with the number of processors used. For example, for class B it went from 1.64% with 2 processors to 17.14% with 32.

Interestingly, manual asynchronous updates reduced execution time only by 0.42, 1.19 and 0.36% with respect to automatic UoR for classes A, B and C, respectively. Here, we noticed that the asynchronous update in fact tended to slow down the execution slightly when more than 8 processors were used, i.e. when some communications took place in the same node. We believe that this is due to the asynchronous communication being more demanding of the OS and the communication layer that the processors in the same node share, thus being partially serialized. Excluding those runs the speedups would have been 1.8, 1.31 and 0.62%, respectively, with the improvement always growing with the number of processors (e.g. from 1.32% with 2 processors to 2.39% with 8 for class B). The main reason why the asynchronous update helps less in this system is the reduction of overlap it enables between computation and communication in this cluster, as the computations are faster, whereas the communications are slower.

4.3. Code readability and programmer productivity

The main motivation for our research is that the traditional shadow region implementations distort the stencil algorithms, because of the complexity of the codes to update them, which often end up being longer than the computational kernels they help. As an example, the NAS MG benchmark used in Section 4.2.1 uses the routine `comm3` to update the shadow regions of most of the stencils of the program. In the original MPI implementation `comm3` and the routines it invokes are 327 lines long, whereas the three stencils that use `comm3` require 96 lines (114 with debugging code): 23 in `resid`, 24 in `psinv` and 49 in `rprj3`. Languages that provide a global view of the data are not much better: the Co-array Fortran [10] version of `comm3` and the routines it invokes need 302 lines. Even in the serial code or in the OpenMP [11], where the shadow regions are necessary to store the periodic boundaries, the amount of extra code is significant. In the serial version of MG, `comm3` has 29 lines, whereas in the OpenMP version, `comm3` has 26 lines. The HTA version without overlapped tiling comes close with 32 lines, but with our proposed overlapped tiling, `comm3` is not necessary at all.

Overlapped tiling helps programming not only by reducing the number of communication statements but also by simplifying indexing in both sequential and parallel programs. For example, operations such as `mapReduce()` or assignment apply only to the owned regions of each tile. Using overlapped tiling, the reference to an HTA `h` means that all the elements in the owned regions participate. On the contrary, the original HTA program has to explicitly index the inner regions for each tile. Table III illustrates the reduction in communication statements and indexing operations for the 3D Stencil, MG and LU benchmarks. It is worth mentioning that the computational part of the 3D Jacobi code uses a specialized stencil function that avoids explicit indexing. If explicit indexing were used in the computational part, the 3D Jacobi without overlapping tiling would have 22 indexing operations, and the one with overlapped tiling, only 8.

We have not gathered extensive quantitative measurements on the learning curve, but an experienced programmer, who had not written parallel programs in the past six years, was able to learn to develop the HTA programs in about one week. He found the overlapped tiling feature very intuitive and he could use it to develop two parallel applications in a straightforward manner. One of them was very similar to the 1D Jacobi shown in Figure 3, and the other was the well-known

Table III. Code difference between original (Orig) and overlapped (OL) HTA programs.

	Benchmark					
Metric	3D Jacobi Orig	3D Jacobi OL	MG Orig	MG OL	LU Orig	LU OL
Communication statements	14	0	117	43	56	4
Indexing operations	12	0	265	117	428	26

Game of Life problem. This is a more complex stencil in which the state of each cell in a bidimensional grid evolves depending on the values of its neighbors in the previous generation. The typical development time for the MPI version of this code is in the range of 10–35 h according to Zelkowitz *et al.* [12]. He developed his version in one working day (about 7 h) and spent another day optimizing it.

5. RELATED WORK

Tiling [1] has been extensively studied both as a way to exploit locality [13] and to enable/improve parallelism [14]. The existence of several studies devoted specifically to tiling for stencil codes, focused on both data locality [15] and parallelism [16], gives an idea of the importance of this optimization in these kinds of codes. Krishnamoorthy *et al.* [16] explore the usage of overlapped tiles as a way to remove inter-tile dependencies.

Regarding the support for overlapped tiles and shadow regions by the programming environment, most approaches have fallen in one of the two extremes. Although most programming approaches that offer a single-threaded view of the computation hide them completely (HPF [17], ZPL [18]), those that require the explicit management of the different threads usually lay this burden on the user (CAF [10], UPC [19]). Thus, the former run the risk of exceeding the compiler capabilities and failing to provide a reasonable performance, whereas the latter reduce programmer productivity with long error-prone codes.

The approaches that fall in between can be classified as either task- or data-centric. The former [20,21] focus on computation partitioning with replicated computation of boundary values, and they require the user to mark each loop nest in which the overlapping is to be applied. Data-centric approaches [3,22,23] are usually library-based and they link the shadow regions to the data structures they extend rather than to regions of code. For example, POOMA [22] allows one to define internal guard layers (shadow regions) between the patches (tiles) that are updated automatically, i.e. their automatic update cannot be disabled. Separate external guard layers (boundaries) can as well be defined, although only some of the containers provided by POOMA allow one to update them automatically. Also, no asynchronous update functions are provided. Still, POOMA's most important difference with respect to HTAs is that its tiles and shadow regions are largely hidden from the user once they are created. This is because POOMA's patches are not used to express parallel computation and communication as HTA tiles do. Instead, POOMA mostly accesses and uses its arrays as a whole unit. Thus, it would be difficult, for example, to write *pipelined* stencil computations as those shown in LU, which need to choose different sets of tiles to express the communication and computations to be performed in each step.

Global arrays [23] (GAs) allow the definition of ghost cells, although their SPMD programming model, the unavailability of automatic updates and asynchronous manual updates, and the (non-object-oriented) interface involving calls to explicit functions to gather information to access the shadow regions make them radically different from POOMA and the HTAs. In addition, GAs are less flexible, as their ghost regions have the same width in the positive and the negative directions in each dimension, and the boundaries are fixed to be periodic.

Finally, it is interesting to notice that, to our knowledge, the HTAs are the only approach that provides support for overlapped tiling in sequential computations, despite the usefulness of this technique to help exploit locality (see Section 4.1). All together, we think that HTAs are the data-centric approach that provides the most convenient interface and control for overlapped tiling.

6. CONCLUSIONS

We have discussed the syntax, implementation details, performance and productivity benefits of the *overlapped tiling* construct in the HTA class. Our experiments show that it greatly simplifies programs not only by automating the otherwise error-prone update of the shadow regions but also by making the indexing simpler and consistent with the non-tiled versions of the codes. As a result, it should have a positive impact on programmer productivity. In addition, its efficient implementation provides little performance degradation compared with hand-optimized codes and in fact its usage speeds up the original HTA codes. A detailed comparison with the previous approaches shows that *overlapped tiling* compares favorably in terms of interface, flexibility and control offered to the programmer.

REFERENCES

1. McKellar AC, Coffman JEG. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM* 1969; **12**(3):153–165.
2. Bikshandi G, Guo J, Hoeflinger D, Almasi G, Fraguela BB, Garzarán MJ, Padua D, von Praun C. Programming for parallelism and locality with hierarchically tiled arrays. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'06*), 2006; 48–57.
3. Bikshandi G, Guo J, von Praun C, Tanase G, Fraguela BB, Garzarán MJ, Padua D, Rauchwerger L. Design and use of htalib—A library for hierarchically tiled arrays. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, November 2006.
4. Guo J, Bikshandi G, Fraguela BB, Garzarán MJ, Padua D. Programming with tiles. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'08*), 2008; accepted for publication.
5. NAS Parallel Benchmarks. Website. http://www.nas.nasa.gov/Software/NPB/.
6. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. *Symposium on Operating System Design and Implementation* (*OSDI*), 2004.
7. Fox GC, Johnson MA, Lyzenga GA, Otto SW, Salmon JK, Walker DW. *Solving Problems on Concurrent Processors. Vol. 1: General Techniques and Regular Problems.* Prentice-Hall: Englewood Cliffs, NJ, 1988.
8. Hiranandani S, Kennedy K, Tseng C-W. Compiler optimizations for Fortran D on MIMD distributed-memory machines. *Proceedings of Supercomputing '91.* ACM Press: New York, 1991; 86–100.
9. Bikshandi G. Parallel programming with hierarchically tiled arrays. *PhD Thesis*, 2007.
10. Numrich RW, Reid J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum* 1998; **17**(2):1–31.
11. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R. *Parallel Programming in OpenMP*. Morgan Kaufmann: San Francisco, CA, U.S.A., 2001.
12. Zelkowitz M, Basili V, Asgari S, Hochstein L, Hollingsworth J, Nakamura T. Measuring productivity on high performance computers. *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium* (*METRICS'05*), 2005; 6.

13. Wolf ME, Lam MS. A data locality optimizing algorithm. *Proceedings of PLDI'91*, 1991; 30–44.
14. Xue J. *Loop Tiling for Parallelism.* Kluwer Academic Publishers: Dordrecht, 2000.
15. Rivera G, Tseng C-W. Tiling optimizations for 3d scientific computations. *Proceedings of Supercomputing '00*, 2000; 32.
16. Krishnamoorthy S, Baskaran M, Bondhugula U, Ramanujam J, Rountev A, Sadayappan P. Effective automatic parallelization of stencil computations. *Proceedings of PLDI 2007*, 2007; 235–244.
17. Koelbel C, Mehrotra P. An overview of high performance Fortran. *SIGPLAN Fortran Forum* 1992; **11**(4):9–16.
18. Chamberlain BL, Choi S, Lewis E, Lin C, Snyder S, Weathersby W. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering* 1998; **5**(3):76–86.
19. Carlson W, Draper J, Culler D, Yelick K, Brooks E, Warren K. Introduction to UPC and language specification. *Technical Report CCS-TR-99-157*, IDA Center for Computing Sciences, 1999.
20. Sawdey A, O'Keefe M. Program analysis of overlap area usage in self-similar parallel programs. *Proceedings of LCPC*, 1997; 79–93.
21. Adve V, Jin G, Mellor-Crummey J, Yi Q. High performance Fortran compilation techniques for parallelizing scientific codes. *Proceedings of Supercomputing '98*. IEEE Computer Society: Silver Spring, MD, 1998; 1–23.
22. Reynders JVW, Hinker PJ, Cummings JC, Atlas SR, Banerjee S, Humphrey WF, Sin SRK, Keahey K, Srikant M, Tholburn MD. POOMA: A framework for scientific simulations of parallel architectures. *Parallel Programming in C++*. MIT Press: Cambridge, MA, 1996; 547–588.
23. Nieplocha J, Krishnan M, Palmer B, Tipparaju V, Ju J. The Global Arrays User's Manual. 2006.