Multidimensional Blocking in UPC

Christopher Barton[1], Călin Caşcaval[2], George Almasi[2], Rahul Garg[1], José Nelson Amaral[1], and Montse Farreras[3]

[1] University of Alberta, Edmonton, Canada
[2] IBM T.J. Watson Research Center
[3] Universitat Politècnica de Catalunya

Abstract. Partitioned Global Address Space (PGAS) languages offer an attractive, high-productivity programming model for programming large-scale parallel machines. PGAS languages, such as Unified Parallel C (UPC), combine the simplicity of shared-memory programming with the efficiency of the message-passing paradigm by allowing users control over the data layout. PGAS languages distinguish between private, shared-local, and shared-remote memory, with shared-remote accesses typically much more expensive than shared-local and private accesses, especially on distributed memory machines where shared-remote access implies communication over a network.
In this paper we present a simple extension to the UPC language that allows the programmer to block shared arrays in multiple dimensions. We claim that this extension allows for better control of locality, and therefore performance, in the language.
We describe an analysis that allows the compiler to distinguish between local shared array accesses and remote shared array accesses. Local shared array accesses are then transformed into direct memory accesses by the compiler, saving the overhead of a locality check at runtime. We present results to show that locality analysis is able to significantly reduce the number of shared accesses.

1 Introduction

Partitioned Global Address Space (PGAS) languages, such as UPC [14], Co-Array Fortran [10], and Titanium [16], extend existing languages (C, Fortran and Java, respectively) with constructs to express parallelism and data distributions. They are based on languages that have a large user base and therefore there is a small learning curve to move codes to these new languages.

We have implemented several parallel algorithms — stencil computation and linear algebra operations such as matrix-vector and Cholesky factorization — in the UPC programming language. During this effort we identified several issues with the current language definition, such as: rudimentary support for data distributions (shared arrays can be distributed only block cyclic), flat threading model (no ability to support subsets of threads), and shortcomings in the collective definition (no collectives on subsets of threads, no shared data allowed as target for collective operations, no concurrent participation of a thread in multiple collectives). In addition, while implementing a compiler and runtime system we found that naively translating all shared accesses to runtime calls is prohibitively expensive. While the language supports block transfers and cast

operations that could alleviate some of the performance issues, it is more convenient to address these problems through compiler optimizations.

Tackling some of these issues, this paper makes the following contributions:

- propose a new data distribution directive, called multidimensional blocking, that allows the programmer to specify n-dimensional tiles for shared data (see Section 2);
- describe a compile-time algorithm to determine the locality of shared array elements and replace references that can be proven to be locally owned by the executing thread with direct memory accesses. This optimization reduces the overhead of shared memory accesses and thus brings single thread performance relatively close to serial implementations, thereby allowing the use of a scalable, heavier, runtime implementation that supports large clusters of SMP machines (see Section 3);
- present several benchmarks that demonstrate the benefits of the multidimensional blocking features and the performance results of the locality analysis; these performance results were obtained on a cluster of SMP machines, which demonstrates that the flat threading model can be mitigated through knowledge in the compiler of the machine architecture (Section 5).

2 Multidimensional Blocking of UPC arrays

In this section we propose an extension to the UPC language syntax to provide additional control over data distribution: tiled (or *multiblocked*) arrays. Tiled data structures are used to enhance locality (and therefore performance) in a wide range of HPC applications [2]. Multiblocked arrays can help UPC programmers to better express these types of applications, allowing the language to fulfill its promise of allowing both high productivity and high performance. Also, having this data structure available in UPC facilitates using library routines, such as BLAS [4], in C or Fortran that already make use of tiled data structures.

Consider a simple stencil computation on a 2 dimensional array that calculates the average of the four immediate neighbors of each element.

```
1 shared double A[M][N];
2 ...
3 for (i=1..M-2,j=1..N-2)
4   B[i][j] = 0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
```

Since it has no data dependencies, this loop can be executed in parallel. However, the naive declaration of `A` above yields suboptimal execution, because e.g. `A[i-1][j]` will likely not be on the same UPC thread as `A[i][j]` and may require inter-node communication to get to. A somewhat better solution allowed by UPC is a striped 2D array distribution:

```
shared double [M*b] A[M][N];
```

$M \times b$ is the *blocking factor* of the array; that is, the array is allocated in contiguous blocks of this size. This however, limits parallelism to $\frac{N}{b}$ processors and causes $O(\frac{1}{b})$ remote array accesses. By contrast, a tiled layout provides $\frac{M \times N}{b^2}$ parallelism and $O(\frac{1}{b^2})$ of the accesses are remote. Typical MPI implementations of stencil computation tile the array and exchange "border regions" between neighbors before each iteration. This approach is also possible in UPC:

```
struct block { double tile[b][b]; };
shared block A[M/b][N/b];
```

However, the declaration above complicates the source code because two levels of indexing are needed for each access. We cannot pretend that `A` is a simple array anymore. We propose a language extension that can declare a tiled layout for a shared array, as follows:

```
shared <type> [b0][b1]...[bn] A[d0][d1] ... [dn];
```

Array `A` is an n-dimensional tiled (or "multi-blocked") array with each tile being an array of dimensions $[b0][b1]...[bn]$. Tiles are understood to be contiguous in memory.

2.1 UPC array layout

To describe the layout of multiblocked arrays in UPC, we first need to discuss conventional shared arrays. A UPC array declared as below:

```
shared [b] <type> A[d0][d1]...[dn];
```

is distributed in memory in a block-cyclic manner with blocking factor `b`. Given an array index $\mathbf{v} = v_0, v_1, ...v_{n-1}$, to locate element $A[\mathbf{v}]$ we first calculate the linearized row-major index (as we would in C):

$$L(\mathbf{v}) = v_0 \times \prod_{j=1}^{n-1} d_j + v_1 \times \prod_{j=2}^{n-1} d_j + ... + v_{n-1} \tag{1}$$

Block-cyclic layout is based on this linearized index. We calculate the UPC *thread* on which array element $A[\mathbf{v}]$ resides. Within the local storage of this thread the array is kept as a collection of blocks. The *course* of an array location is the block number in which the element resides; the *phase* is its location within the block.

$$\begin{cases} thread(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b} \right\rfloor \bmod \mathcal{T} \\ phase(A, \mathbf{v}) & ::= L(\mathbf{v}) \bmod b \\ course(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b \times \mathcal{T}} \right\rfloor \end{cases}$$

Multiblocked arrays: The goal is to extend UPC syntax to declare tiled arrays while minimizing the impact on language semantics. The internal representation of multiblocked arrays should not differ too much from that of standard UPC arrays. Consider a multiblocked array `A` with dimensions $D = \{d_0, d_1, ...d_n\}$ and blocking factors $B = \{b_0, b_1, ...b_n\}$. This array would be allocated in $k = \prod_{i=0}^{n-1} \left\lceil \frac{d_i}{b_i} \right\rceil$ blocks (or tiles) of $b = \prod_{i=0}^{n-1} b_i$ elements. We continue to use the concepts of *thread*, *course* and *phase* to find array elements. However, for multiblocked arrays two linearized indices must be computed: one to find the block and another to find an element's location within a

block. Note the similarity of Equations 2 and 3 to Equation 1:

$$L_{in-block}(\mathbf{v}) = \sum_{k=0}^{n-1}((v_k \bmod b_k) \times \prod_{j=k+1}^{n-1} b_j) \tag{2}$$

$$L(\mathbf{v}) \quad = \sum_{k=0}^{n-1}(\left\lfloor \frac{v_k}{b_k} \right\rfloor \times \prod_{j=k+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil) \tag{3}$$

The *phase* of a multiblocked array element is its linearized in-block index. The *course* and *thread* are calculated with a cyclic distribution of the block index, as in the case of regular UPC arrays.

$$\begin{cases} thread(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i} \right\rfloor \bmod \mathcal{T} \\ phase(A, \mathbf{v}) & ::= L_{in-block}(\mathbf{v}) \\ course(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i \times \mathcal{T}} \right\rfloor \end{cases} \tag{4}$$

Array sizes that are non-multiples of blocking factors: The blocking factors of multiblocked arrays are not required to divide their respective dimensions, just as blocking factors of regular UPC arrays are not required to divide the array's dimension(s). Such arrays are padded in every dimension to allow for correct index calculation.

2.2 Multiblocked arrays and UPC pointer arithmetic

The address of any UPC array element (even remote ones) can be taken with the `upc_addressof` function or with the familiar `&` operator. The result is called a *pointer-to-shared*, and it is a reference to a memory location somewhere within the space of the running UPC application. In our implementation a pointer-to-shared identifies the base array as well as the thread, course and phase of an element in that array.

UPC pointers-to-shared behave much like pointers in C. They can be incremented, dereferenced, compared etc. The familiar pointer operators (`*`, `&`, `++`) are available. A series of increments on a pointer-to-shared will cause it to traverse a UPC shared array in row-major order.

Pointers-to-shared can also used to point to multiblocked arrays. Users can expect pointer arithmetic and operators to work on multiblocked arrays just like on regular UPC shared arrays.

Affinity, casting and dynamic allocation of multiblocked arrays: Multiblocked arrays can support affinity tests (similar to the `upc_threadof` function) and type casts the same way regular UPC arrays do.

Dynamic allocation of UPC shared arrays can also be extended to multiblocked arrays. UPC primitives like `upc_all_alloc` always return shared variables of type `shared void *`; multiblocked arrays can be allocated with such primitives as long as they are cast to the proper type.

2.3 Implementation Issues

Pointers and dynamic allocation of arrays: Our current implementation supports only statically allocated multiblocked arrays. Dynamically allocated multiblocked arrays could be obtained by casting dynamically allocated data to a shared multiblocked type, making dynamic multiblocked arrays a function of correct casting and multiblocked pointer arithmetic. While correct multiblocked pointer arithmetic is not conceptually difficult, implementation is not simple: to traverse a multiblocked array correctly, a pointer-to-shared will have to have access to all blocking factors of the shared type.
Processor tiling: Another limitation of the current implementation is related to the cyclic distribution of blocks over UPC threads. An alternative would be to specify a processor grid to distribute blocks over. Equation 3 would have to be suitably modified to take thread distribution into consideration. We have not implemented this yet in the UPC runtime system, although performance results presented later in the paper clearly show the need for it.
Hybrid memory layout: Our UPC runtime implementation is capable of running in mixed multithreaded/multinode environments. In such an environment locality is interpreted on a per-node basis, but array layouts have to be on a per-UPC-thread basis to be compatible with the specification. This is true both for regular and multiblocked arrays.

3 Locality Analysis for Multi-Dimensional Blocking Factors

This section describes a compile-time analysis for multi-dimensional blocking factors in UPC shared arrays. The analysis considers loop nests that contain accesses to UPC shared arrays and finds shared array references that are provably local (on the same UPC thread) or shared local (on the same node in shared memory, but on different UPC threads). All other shared array references are potentially remote (reachable only via inter-node communication).

The analysis enables the compiler to refactor the loop nest to separate local and remote accesses. Local and shared local accesses cause the compiler to generate simple memory references; remote variable accesses are resolved through the runtime with a significant remote access overhead. We consider locality analysis crucial to obtaining good performance with UPC.

In Figure 1 we present a loop nest that will be used as an example for our analysis. In this form the shared array element in the affinity test — the last parameter in the `upc_forall` statement — is formed by the current loop-nest index, while the single element referenced in the loop body has a displacement, with respect to the affinity expression, specified by the distance vector $\mathbf{k} = [k_0, k_1, \ldots, k_{n-1}]$. Any loop nest in which the index for each dimension, both in the affinity test and in the array reference, is an affine expression containing only the index in the corresponding dimension can be transformed to this cannonical form.[4] Table 1 summarizes the notation used throughout this section and the expressions used by the locality analysis to compute the locality of

[4] An example of a loop nest that cannot be transformed to this cannonical form is a two-level nest accessing a two-dimensional array in which either the affinity test or the reference contains an expression such as `A[`$v_0 + v_1$`][`v_1`]`.

```
shared [b0][b1]···[bk−1] int A[d0][d1]···[dk−1];
for(v0=0 ; v0 < d0 − k0 ; v0++)
    for(v1=0 ; v1 < d1 − k1 ; v1++){
        ...
        upc_forall(vn−1=0 ; vn−1 < dn−1 − kn−1 ; vn−1++ ; &A[v0][v1]...[vn−1])
                A[v0 + k0][v1 + k1]...[vn−1 + kn−1] = v0 ∗ v1 ∗ ... ∗ vn−1;
    }
```

Fig. 1. Multi-level loop nest that accesses a multi-dimensional array in UPC.

Ref	Expression	Description
1	n	number of dimensions
2	b_i	blocking factor in dimension i
3	d_i	array size in dimension i
4	v_i	position index in dimension i
5	$\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$	Index of an array element
6	$\mathcal{T}$	number of threads
7	t	number of threads per node
8	$\mathcal{B}_i = \lfloor \frac{v_i}{b_i} \rfloor$	Block index in dimension i
9	$L(\mathbf{v}) = \sum_{i=0}^{n-1} \mathcal{B}_i \times \prod_{j=i+1}^{n-1} \lceil \frac{d_j}{b_j} \rceil$	Linearized block index
10	$L'(\mathbf{v}) = L(\mathbf{v}) \% \mathcal{T}$	Normalized linearized block index
11	$\mathcal{N}(\mathbf{v}) = \lfloor \frac{L'(\mathbf{v})}{t} \rfloor$	Node ID
12	$\mathcal{O}(\mathbf{v}) = L(\mathbf{v}) \% t$	Block offset within a node

Table 1. Expressions used to compute the node ID that each element $A[\mathbf{v}]$ of array A belongs to.

array elements. The goal of the locality analysis is to compute symbolically the node ID of each shared reference in the loop and compare it to the node ID of the affinity expression. All references having a node ID equal to the affinity expression's node ID are local.

Locality analysis is done on the n-dimensional blocks of the multiblocked arrays present in the loop. For conventional UPC shared arrays declared with a blocking factor b, the analysis uses blocking factors of 1 in all dimensions except the last dimension, where b is used. The insight of the analysis is that a block shifted by a displacement vector $\mathbf{k}$ can span at most two threads along each dimension. Therefore locality can only change in one place in this dimension. We call this place the cut.

Once the cut is determined, our analysis tests the locality of the elements at the 2^n corners of the block. If a corner is found to be local, all the elements in the region from the corner up to the cuts in each dimension are also local.

Definition 1. *The value of a cut in dimension i, Cut_i, is the distance, measured in number of elements, between the corner of a block and the first transition between nodes on that dimension.*

Consider the two-level loop nest that accesses a two-dimensional blocked array shown in Figure 2. The layout of the array used in the loop is shown to the right of the code. Thin lines separate the elements of the array. Large bold numbers inside each block of 2×3 elements denote the node ID to which the block is mapped. Thick lines separate nodes from each other. The grey area in the array represents all elements that are ref-

```
1 /* 8 threads and 2 threads/node */
2
3 shared [2][3] int A[8][8];
4
5 for(v0=0; v0<7 ; v0++){
6   upc_forall(v1=0; v1<6 ; v1++; &A[v0][v1]){
7     A[v0+1][v1+2] = v0*v1;
8   }
9 }
```

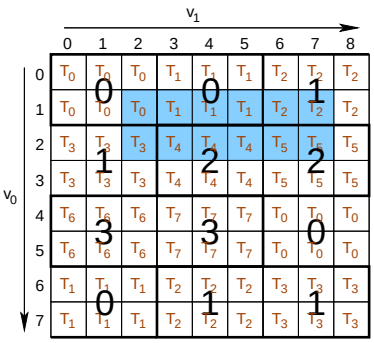

Fig. 2. A two-dimensional array node example.

erenced by iterations of the `forall` loop that are affine with `&A[0][0]`; cuts in this iteration space are determined by the thick lines (node boundaries).

Finding the cuts: In general, for dimensions $i = 0$ to $n-2$ the value of the cut in that dimension is given by the following expression.

$$\mathrm{Cut}_i = b_i - k_i \,\%\, b_i \tag{5}$$

Thus in the example $\mathrm{Cut}_0 = 1$, which means that the only possible change of locality value happens between the first and second row of the block being accessed.

The cuts in the last dimension of the array are not necessarily the same for each corner. In Figure 2, for the top corners the cut is 4 but for the bottom corners the cut is 1. This happens when there are multiple colocated UPC threads in a node (in a hybrid setup); because the blocks in a node may "wrap around" the rows in the array.

Thus the analysis has to compute two separate values for the last cut: one for the upper corners and a second one for the lower corners. Upper and Lower refers to the last dimension in a multi-dimensional array. Let $\mathbf{k}' = [k_0 + b_0 - 1, k_1, \ldots, k_{n-1}]$. The expression for the last cut in the upper corner is as follows:

$$\mathrm{Cut}_{n-1}^{\mathrm{Upper}} = (t - \mathcal{O}(\mathbf{k})) \times b_{n-1} - k_{n-1} \,\%\, b_{n-1} \tag{6}$$

$$\mathrm{Cut}_{n-1}^{\mathrm{Lower}} = (t - \mathcal{O}(\mathbf{k}')) \times b_{n-1} - k_{n-1} \,\%\, b_{n-1} \tag{7}$$

where t is the number of threads per node.

When there is a single thread per node (*i.e.* $t = 1$), the normalized linearized block index is zero, and thus equations 6 and 7 simplify to equation 5.

Axiom 3.01 *Given an* `upc_forall` *loop with affinity test* $\mathrm{AffTest} = A(\mathbf{v})$ *and a shared array reference* $\mathrm{Ref} = A(\mathbf{v} + \mathbf{k})$*, this reference is local if and only if* $\mathcal{N}(\mathrm{AffTest}) = \mathcal{N}(\mathrm{Ref})$

Theorem 1. *Let A be an n-dimensional shared array with dimensions $d_0, d_1, \ldots, d_{n-1}$ and with blocking dimensions $b_0, b_1, \ldots, b_{n-1}$. Let $\mathbf{w} = v_0, v_1, \ldots, v_p, \ldots, v_{n-1}$ and $\mathbf{y} = v_0, v_1, \ldots, v_p + 1, \ldots v_{n-1}$ be two vectors such that $A(\mathbf{w})$ and $A(\mathbf{y})$ are elements of A. Let $B_{\mathrm{Off}} = \mathcal{O}(v_0, v_1, \ldots, 0)$ be the block offset for the first element in the block in dimension $n-1$. Let*

$$v_i' = \begin{cases} v_i \,\%\, b_i - k_i \,\%\, b_i & \text{if} \quad i \neq n-1 \\ (v_i + B_{\mathrm{Off}} \times b_i) \,\%\, (b_i \times t) & \text{Otherwise.} \end{cases} \tag{8}$$

if $v'_p \neq \mathrm{Cut}_p - 1$ *then* $\mathcal{N}(\mathbf{w}) = \mathcal{N}(\mathbf{y})$.

Proof. We only present the proof for the case $p \neq n-1$ here. The proof for the case $p = n-1$ follows a similar reasoning but is more involved because it has to take into account the block offset for the first element in dimension $n-1$.
From the expressions in Table 1 the expression for the node id of elements $\mathbf{w}$ and $\mathbf{y}$ are given by:

$$\mathcal{N}(\mathbf{w}) = \left\lfloor \frac{L(\mathbf{w}) \% \mathcal{T}}{t} \right\rfloor \text{ and } \mathcal{N}(\mathbf{y}) = \left\lfloor \frac{L(\mathbf{y}) \% \mathcal{T}}{t} \right\rfloor \tag{9}$$

The linearized block index for $\mathbf{w}$ and $\mathbf{y}$ can be written as:

$$L(\mathbf{w}) = \sum_{i=0}^{n-1} \left\lfloor \frac{v_i}{b_i} \right\rfloor \times \prod_{j=i+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \tag{10}$$

$$L(\mathbf{y}) = L(\mathbf{w}) + \left(\left\lfloor \frac{v_p + 1}{b_p} \right\rfloor - \left\lfloor \frac{v_p}{b_p} \right\rfloor \right) \times \prod_{j=p+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \tag{11}$$

From equations 5 and 8:

$$v'_p = \mathrm{Cut}_p - 1 \tag{12}$$

$$v_p \% \ b_p - k_p \% \ b_p = b_p - k_p \% \ b_p - 1 \tag{13}$$

From equation 13, the condition $v'_p \neq \mathrm{Cut}_p - 1$ implies that $v_p \% \ b_p \neq b_p - 1$, which implies that $v_p \% \ b_p \leq b_p - 2$. Therefore:

$$\left\lfloor \frac{v_p + 1}{b_p} \right\rfloor = \left\lfloor \frac{v_p}{b_p} \right\rfloor \tag{14}$$

Substituting this result in equation 11 results that $L(\mathbf{y}) = L(\mathbf{w})$ and therefore $N(\mathbf{w}) = N(\mathbf{y})$.

Theorem 1 is the theoretical foundation of locality analysis based on corners and cuts. It establishes that the only place within a block where the node ID may change is at the cut. The key is that the elements $A(\mathbf{w})$ and $A(\mathbf{y})$ are adjacent elements of A.

4 Identifying Local Shared Accesses

In this section we present an algorithm that splits a loop nest into a number of smaller regions in the iteration space, such that in each region, each shared reference is known to be local or known to be remote. In a region, if a shared reference is determined to be local then the reference is privatized otherwise a call to the runtime is inserted.

To determine such regions, our analysis reasons about the positions of various shared references occuring in the loop nest relative to the affinity test expression. For each region, we keep track of a *position* relative to the affinity test shared reference. For each shared reference in the region, we also keep track of position of each reference relative to the region.

We start with the original loop nest as a single region. This region is analyzed and the cuts are computed. The region is then split according to the cuts generated. The new generated regions are again analyzed and split recursively until no more cuts are required. When all of the regions have been generated, we use the position of the region, and the position of the shared reference within the region to determine if it is local or remote. All shared references that are local are privatized. Figure 3 provides a sample

```
1 shared [5][5] int A[20][20];
2 int main() {
3   int i,j;
4   for (i=0; i < 19; i++)
5     upc_forall(j=0; j < 20; j++; &A[i][j]) {
6       A[i+1][j] = MYTHREAD;
7     }
8 }
```

Fig. 3. Example `upc_forall` loop containing a shared reference

loop nest containing a `upc_forall` loop and a shared array access. We will assume the example is compiled for a machine containing 2 nodes and will run with 8 UPC threads, creating a thread group size of 4. In this scenario, the shared array access on Line 6 will be local for the first four rows of every block owned by a thread T and remote for the remaining row. The LOCALITYANALYSIS algorithm in Figure 4 begins by collecting all top-level loop nests that contain a candidate `upc_forall` loop. To be a candidate for locality analysis, a `upc_forall` loop must be normalized (lower bound begins at 0 and the increment is 1) and must use a pointer-to-shared argument for the affinity test. The algorithm then proceeds to analyze each loop nest independently (Step 2).

Phase 1 of the per-loopnest analysis algorithm finds and collects the `upc_forall` loop l_{forall}. The affinity statement used in l_{forall}, A_{stmt} is also obtained. Finally the COLLECTSHAREDREFERENCES procedure collects all candidate shared references in the specified `upc_forall` loop. In order to be a candidate for locality analysis, a shared reference must have the same blocking factor as the shared reference used in the affinity test. The compiler must also be able to compute the *displacement vector* $k = ref_{shared} - affinityStatement$ for the shared reference, the vectorized difference between the indices of the reference and of the affinity statement.

In the example in Figure 3 the loop nest on Line 4 is collected as a candidate for locality analysis. The shared reference on Line 6 is collected as a candidate for locality analysis; the computed displacement vector is [1,0].

Phase 2 of the algorithm restructures the loop nest by splitting the iteration space of each loop into *regions* where the locality of shared references is known. Each region has a *statement list* associated with it, i.e. the lexicographically ordered list of statements as they appear in the program. Each region is also associated with a *position* in the iteration space of the loops containing the region.

In the example in Figure 3 the first region, R_0 contains the statements on Lines 5 to 7. The position of R_0 is 0, since the iteration space of the outermost loop contains the location 0. Once initialized, the region is placed into a list of regions, $\mathcal{L}_R$ (Step 8).

The algorithm iterates through all regions in $\mathcal{L}_R$. For each region, a list of cuts is computed based on the shared references collected in Phase 1. The cut represents the

```
LocalityAnalysis(Procedurep)
1. NestSet ← GatherForallLoopNests(p)
2. foreach loop nest L in NestSet
Phase 1 - Gather Candidate Shared References
3.      l_forall ← upc_forall loop found in loop nest L
4.      nestDepth ← depth of L
5.      A_stmt ← Affinity statement used in l_forall
6.      SharedRefList ← CollectSharedReferences(l_forall, A_stmt)
Phase 2 - Restructure Loop Nest
7.      FirstRegion ← InitializeRegion(L)
8.      L_R ← FirstRegion
9.      while L_R not empty
10.         R ← Pop head of L_R
11.         CutList ← GenerateCutList(R, SharedRefList)
12.         nestLevel ← R.nestLevel
13.         if nestLevel < nestDepth − 1
14.           L_R ← L_R ∪ GenerateNewRegions(R, CutList)
15.         else
16.           L_R^final ← L_R^final ∪ GenerateNewRegions(R, CutList)
17.         endif
18.     end while
Phase 3 - Identify Local Accesses and Privatize
19.     foreach R in L_R^final
20.         foreach ref_shared in SharedRefList
21.             refPosition ← ComputePosition(ref_shared, R)
22.             nodeId ← ComputeNodeId(ref_shared, refPosition)
23.             if nodeId = 0
24.               PrivatizeSharedReference(ref_shared)
25. endfor
```

Fig. 4. Locality analysis for UPC shared references

transition between a local access and a remote access in the given region. The GENERATECUTLIST algorithm first determines the loop-variant induction variable iv in R that is used in ref_{shared}. The use of iv identifies the dimension in which to obtain the blocking factor and displacement when computing the cut. Depending on the dimension of the induction variable, either Equation 5 or Equations 6 and 7 are used to compute the cuts.

GENERATECUTLIST sorts all cuts in ascending order. Duplicate cuts and cuts outside the iteration space of the region ($Cut = 0$ or $Cut \geq b$) are discarded. Finally, the current region is cut into multiple iteration ranges, based on the cut list, using the GENERATENEWREGION algorithm. Newly created regions are separated by an if statement containing a *cut expression* of the form $iv\%b < Cut$ (the modulo is necessary since a cut is always in the middle of a block).

Step 13 determines if the region R is located in the innermost loop in the current loop nest (*i.e.* there are no other loops inside of R). If R contains innermost statements the regions generated by GENERATENEWREGIONS are placed in a separate list of final regions. $\mathcal{L}_R^{final}$. This ensures that at the end of Phase 2, the loop nest has been refactored into several iteration ranges and final statement lists (representing the innermost loops) are collected for use in Phase 3.

```
1  shared [5][5] int A[20][20];
2
3  int main() {
4    int i,j;
5    for (i=0; i < 19; i++)
6      if ((i % 5) < 4) {
7        upc_forall(j=0; j < 20; j++;
8                   &A[i][j]) {
9          A[i+1][j] = MYTHREAD;
10       }
11     }
12     else {
13       upc_forall(j=0; j < 20; j++;
14                  &A[i][j]) {
15         A[i+1][j] = MYTHREAD;
16       }
17     }
18 }
```

Fig. 5. Example after first cut

```
1  shared [5][5] int A[20][20];
2  int main() {
3    int i,j;
4    for (i=0; i < 19; i++)
5      if ((i % 5) < 4) {
6        upc_forall(j=0; j < 20; j++;
7                   &A[i][j]) {
8          offset = ComputeOffset(i,j);
9          base_A+offset = MYTHREAD;
10       }
11     }
12     else {
13       upc_forall(j=0; j < 20; j++;
14                  &A[i][j]) {
15         A[i+1][j] = MYTHREAD;
16       }
17     }
18 }
```

Fig. 6. Example after final code generation

The second phase iterates through the example in Figure 3 three times. The first region, R_0 and the $CutList = 4$, calculated by GENERATECUTLIST are passed in and the intermediate code shown in Figure 5 is generated. GENERATENEWREGIONS inserts the `if ((i \% 5) < 4)` branch and replicates the statements in region R_0. Two new regions, R_1 containing statements between lines 8 to 10 and R_2, containing lines 14 to 16, are created and added to the $NewList$. The respective positions associated with R_1 and R_2 are [0] and [4], respectively.

The new regions, R_1 and R_2 are popped off of the region list $\mathcal{L}_R$ in order. Neither region requires any cuts. GENERATENEWREGIONS copies R_1 and R_2 into R_3 and R_4 respectively. Since R_1 and R_2 represent the innermost loops in the nest, the new regions R_3 and R_4 will be placed into the final regions list (Step 16 in Figure 4). The position of region R_3 is [0,0] and the position of region R_4 is [4,0].

Phase 3 of the algorithm uses the position information stored in each of the final regions to compute the position of each shared reference in that region (Step 21). This information is then used to compute the node ID of the shared reference using the equations presented in Section 3 (Step 22). All shared references with a node ID of 0 are local and are privatized (Step 24). The shared reference $ref_{shared}^{R_3}$ located in R_3 is computed to have a position of $[1,0]$ based on the position of R_3,$[0,0]$, and the displacement vector of ref_{shared}, $[1,0]$. The node ID for this position is 0 and thus $ref_{shared}^{R_3}$ is local. The shared reference $ref_{shared}^{R_4}$ is computed to have a position of $[5,0]$ using the position for region R_4, $[4,0]$. The node ID for this position is 1, and thus this reference is remote. Figure 6 shows the final code that is generated.

5 Experimental evaluation

In this section we propose to evaluate the claims we have made in the paper: namely the usefulness of multiblocking and locality analysis. For our evaluation platform we used 4 nodes of an IBM Squadron™cluster. Each node has 8 SMP Power5 processors running at 1.9 GHz and 16 GBytes of memory.

Cholesky factorization and Matrix multiply: Cholesky factorization was written to showcase multi-blocked arrays. The tiled layout allows our implementation to take direct advantage of the ESSL [5] library. The code is patterned after the LAPACK [4] `dpotrf` implementation and adds up to 53 lines of text. To illustrate the compactness

of the code, we reproduce one of the two subroutines used, distributed symmetric rank-k update, below.

```
1  void update_mb (shared double [B][B] A[N][N], int col0, int col1) {
2    double a_local[B*B], b_local[B*B];
3    upc_forall (int ii=col1; ii<N; ii+=B; continue)
4      upc_forall (int jj=col1; jj<ii+B; jj+=B; &A[ii][jj]) {
5        upc_memget (a_local, &A[ii][col0], sizeof(double)*B*B);
6        upc_memget (b_local, &A[jj][col0], sizeof(double)*B*B);
7        dgemm ("T", "N", &n, &m, &p, &alpha, b_local, &B, a_local,
8                &B, &beta, (void *)&A[ii][jj], &B);
9      }
10 }
```

The matrix multiply benchmark is written in a very similar fashion. It amounts to little more than a (serial) `k` loop around the `update` function above with slightly different loop bounds and three shared array arguments `A`, `B` and `C` instead of only one. It amounts to 20 lines of code. Without question, multiblocking allows compact code

Cholesky Performance (GFlops)

	1 node	2 nodes	3 nodes	4 nodes
1 TPN	5.37	10.11	15.43	19.63
2 TPN	9.62	16.19	28.64	35.41
4 TPN	14.98	23.03	45.43	59.14
6 TPN	18.73	35.29	52.57	57.8
8 TPN	26.65	23.55	59.83	74.14

Matrix Multiply Performance (GFlops)

	1 node	2 nodes	3 nodes	4 nodes
1 TPN	5.94	11.30	16.17	22.24
2 TPN	11.76	21.41	29.82	42.20
4 TPN	23.24	39.18	51.05	73.44
6 TPN	31.19	54.51	66.17	89.55
8 TPN	44.20	63.24	79.00	99.71

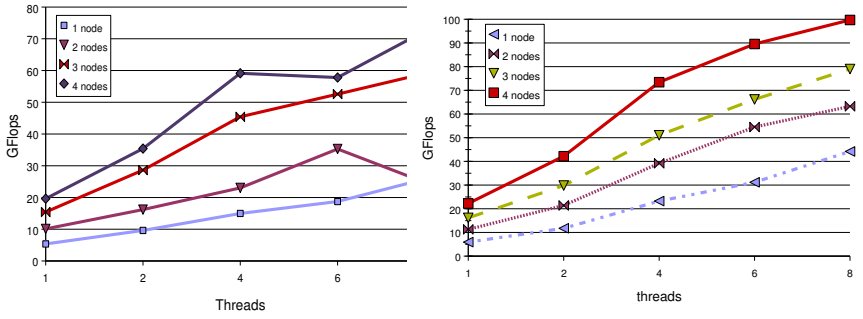

Fig. 7. Performance of multiblocked Cholesky and matrix multiply as a function of participating nodes and threads per node (TPN). Theoretical peak: 6.9 $GFlops \times threads \times nodes$

representation. The benchmark numbers presented in Figures 7 show mediocre scaling and performance "hiccups", which we attribute to communication overhead and poor communication patterns. Clearly, multiblocking syntax needs to be extended with a distribution directive. Also, the UPC language could use better collective communication primitives; but that is in the scope of a future paper.

Dense matrix-vector multiplication: This benchmark multiplies a two-dimensional shared matrix with a one-dimensional shared vector and places the result in a one-

Matrix-vector multiply

Naive	1 node	2 nodes	3 nodes	4 nodes
1 TPN	27.55	16.57	14.13	9.21
2 TPN	16.57	8.59	7.22	4.32
4 TPN	8.57	4.3	3.63	2.18
6 TPN	7.2	3.62	2.43	1.89
8 TPN	4.33	2.2	1.96	1.28

Opt.	1 node	2 nodes	3 nodes	4 nodes
1 TPN	2.08	1.22	0.78	0.6
2 TPN	1.7	0.85	0.63	0.43
4 TPN	0.85	0.44	0.33	0.23
6 TPN	0.65	0.35	0.25	0.19
8 TPN	0.44	0.23	0.22	0.17

Stencil benchmark

Naive	1 node	2 nodes	3 nodes	4 nodes
1 thread	35.64	24.59	19.04	13.41
2 threads	18.85	13.56	9.82	7.9
4 threads	9.8	13.64	5.58	8.9
6 threads	10.85	8.98	7.53	6.12
8 threads	4.9	5.58	9.52	3.66

Opt.	1 node	2 nodes	3 nodes	4 nodes
1 thread	0.30	1.10	1.41	0.74
2 threads	0.73	0.72	0.75	1.06
4 threads	0.44	1.19	0.39	0.84
6 threads	0.32	0.30	1.11	0.75
8 threads	0.22	0.63	1.07	1.02

Fig. 8. Runtime in seconds for the matrix-vector multiplication benchmark (left) and for the stencil benchmark (right). The tables on the top show naive execution times; the tables on the bottom reflect compiler-optimized runtimes.

dimensional shared vector. The objective of this benchmark is to measure the speed difference between compiler-privatized and unprivatized accesses.

The matrix, declared of size 14400×14400, the vector as well the result vector are all blocked using single dimensional blocking. The blocking factors are equivalent to the [*] declarations. Since the vector is shared, the entire vector is first copied into a local buffer using `upc_memget`. The matrix-vector multiplication itself is a simple 2 level nest with the outer loop being `upc_forall` . The address of the result vector element is used as the affinity test expression.

Results presented in Figure 8 (left side) confirm that compiler-privatized accesses are about an order of magnitude faster than unprivatized accesses.

5-point Stencil: This benchmark computes the average of a 4 immediate neighbors and the point itself at every point in a 2 dimensional matrix and stores the result in a different matrix of same size. The benchmark requires one original data matrix and one result matrix. 2-d blocking was used to maximize the locality. The matrix size used for the experiments was 5760×5760. Results, presented in Figure 8 (right side), show that in this case, too, run time is substantially reduced by privatization.

6 Related Work

There is a significant body of work on data distributions in the context of High Performance Fortran (HPF) and other data parallel languages. Numerous researchers have tackled the issue of optimizing communication on distributed memory architectures by either finding an appropriate distribution onto processors [1, 9] or by determining a computation schedule that minimizes the number of message transfers [7, 12]. By contrast to these works, we do not try to optimize the communication, but rather allow the programmer to specify at very high level an appropriate distribution and then eliminate the need for communication all together using compiler analysis. We do not attempt to restructure or improve the data placement of threads to processors in order to minimize communication. While these optimizations are certainly possible in our compiler, we leave them as future work.

The locality analysis presented in this paper is also similar to array privatization [13, 11]. However, array privatization relies on the compiler to provide local copies and/or copy-in and copy-out semantics for all privatized elements. In our approach, once ownership is determined, private elements are directly accessed. In future work we will determine if there is sufficient reuse in UPC programs to overcome the cost of copying array elements into private memory.

Tiled and block distributions are useful for many linear algebra and scientific codes [2]. HPF-1 provided the ability to choose a data distribution independently in each dimesion if desired. Beside HPF, several other languages, such as ZPL [3] and X10 [15] provide them as standard distributions supported by the language. In addition, libraries such as the Hierarichical Tiled Arrays library [2] provide tiled distributions for data decomposition. ScaLAPACK [6], a widely used parallel library provides a 2 dimensional block-cyclic distribution for matrices which allows the placement of blocks over a 2-dimensional processor grid. The distribution used by ScaLAPACK is therefore more general than the distribution presented in the this paper.

7 Conclusions and Future Work

In this paper we presented a language extension for UPC shared arrays that provides fine control over array data layout. This extensions allows the programmer to obtain better performance while simplifying the expression of computations, in particular matrix computations. An added benefit is the ability to integrate existing libraries written in C and Fortran, which require specific memory layouts. We also presented a compile-time analysis and optimization of shared memory accesses. Using this analysis, the compiler is able to reduce the overheads introduced by the runtime system.

A number of issues still remain to be resolved, both in the UPC language and more importantly in our implementation. For multiblocked arrays, we believe that adding processor tiling will increase the programmer's ability to write codes that scale to large numbers of processors. Defining a set of collectives that are optimized for the UPC programming model will also address several scalability issues, such as the ones occuring in the LU Factorization and the High Performance Linpack kernel [8].

Our current compiler implementation suffers from several shortcomings. In particular, several loop optimizations are disabled in the presence of upc_forall loops. These limitations are reflected in the results presented in this paper, where the baseline C compiler offers a higher single thread performance compared to the UPC compiler.

Acknowledgements

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. We also want to thank Philip Luk and Ettore Tiotto for their help with the IBM xlUPC compiler.

References

1. E. Ayguade, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In *Languages and Compilers for Parallel Computing*, pages 61–75, 1994.
2. G. Bikshandi, J. Guo, D. Hoeflinger, G. Almási, B. B. Fraguela, M. J. Garzarán, D. A. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPOPP*, pages 48–57, 2006.
3. B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.
4. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.
5. ESSL User Guide. http://www-03.ibm.com/systems/p/software/essl.html.
6. L. S. B. et al. ScaLAPACK: a linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, page 15 (electronic), Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.
7. M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs:. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, 1996.
8. HPL Algorithm description. http://www.netlib.org/benchmark/hpl/algorithm.html.
9. U. Kremer. Automatic data layout for distributed memory machines. Technical Report TR96-261, 14, 1996.
10. R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.
11. Y. Paek, A. G. Navarro, E. L. Zapata, and D. A. Padua. Parallelization of benchmarks for scalable shared-memory multiprocessors. In *IEEE PACT*, pages 401–, 1998.
12. R. Ponnusamy, J. H. Saltz, A. N. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, 1995.
13. P. Tu and D. A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.
14. *UPC Language Specification, V1.2*, May 2005.
15. The X10 programming language. http://x10.sourceforge.net, 2004.
16. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998.