Combining Static and Dynamic Data Coalescing in Unified Parallel C

Michail Alvanos, *Student Member, IEEE*, Montse Farreras, Ettore Tiotto, José Nelson Amaral, *Senior Member, IEEE*, and Xavier Martorell, *Member, IEEE*

Abstract—Significant progress has been made in the development of programming languages and tools that are suitable for hybrid computer architectures that group several shared-memory multicores interconnected through a network. This paper addresses important limitations in the code generation for Partitioned Global Address Space (PGAS) languages. These languages allow fine-grained communication and lead to programs that perform many fine-grained accesses to data. When the data is distributed to remote computing nodes, code transformations are required to prevent performance degradation. Until now code transformations to PGAS programs have been restricted to the cases where both the physical mapping of the data or the number of processing nodes are known at compilation time. In this paper, a novel application of the inspector-executor model overcomes these limitations and allows profitable code transformations, which result in fewer and larger messages sent through the network, when neither the data mapping nor the number of processing nodes are known at compilation time. A performance evaluation reports both scaling and absolute performance numbers on up to 32768 cores of a Power 775 supercomputer. This evaluation indicates that the compiler transformation results in speedups between 1.15X and 21X over a baseline and that these automated transformations achieve up to 63% the performance of the MPI versions.

Index Terms—Unified Parallel C, Partitioned Global Address Space, One-Sided Communication, Performance Evaluation

1 Introduction

Maintaining the productivity of software with the growing complexity of parallel systems, is a significant concern for the developers. Parallel languages and programming models must provide simple means for developing applications that can run on parallel systems without sacrificing performance. A popular programming model for distributed systems is the Software Distributed Shared Memory systems (DSMs) [1], [2], [3]. Distributed Shared Memory (DSM) systems refer to a wide class of software and hardware implementations, in which each node of a cluster has access to shared memory in addition to each node's non-shared private memory. However, most of the software DSM systems rely on the page-fault mechanism with page prefetching and often have poor performance on fine-grained communication [4]. PGAS languages, such as Unified Parallel C [5], Co-Array Fortran [6], Fortress [7], Chapel [8], X10 [9], and Titanium [10], extend existing languages with constructs to express parallelism and data distribution. These languages provide a shared-memory-like programming model, where the address space is partitioned and the programmer has control over the data layout.

Despite significant effort by the research community to make PGAS languages practical for parallel programming, the de facto programming model for distributed memory architectures is still the Message Passing Interface (MPI) [11]. One reason is that PGAS programs deliver scalable performance only when they are carefully tuned. Often, after initial coding, the programmer tunes the source code to produce a more scalable version. However, the reality is that, at the end of these modifications, the PGAS code resembles very much its MPI equivalent, often nullifying the ease-of-coding advantage of these languages. In the UPC language, the program accesses data using individual reads and writes to the shared space. In a distributed environment, this coding style translates into fine-grained communication, which has poor efficiency and hinders performance of PGAS applications [12].

To cope with the fine-grained communication that arises with the straightforward translation of fine-grained accesses to shared data on a distributed memory system, the research community proposed the coalescing of shared accesses to improve performance. However, existing solutions [12], [13], [14] have two important limitations. (i) They require the knowledge of physical data mapping at compile time. The programmer must specify the number of threads, the number of processing nodes, and the data distribution at compile time. (ii) The compiler can optimize shared

- *Michail Alvanos, Montse Farreras, and Xavier Martorell are with the Barcelona Supercomputing Center, and the Universitat Politecnica de Catalunya, 08034 Barcelona, Spain.
E-mail: malvanos@gmail.com, mfarrera,xavim@ac.upc.edu*
- *Ettore Tiotto is with IBM Toronto Laboratory, Toronto, Canada.
E-mail: etiotto@ca.ibm.com*
- *José Nelson Amaral is with Department of Computing Science, University of Alberta, Edmonton, Canada.
E-mail: amaral@cs.ualberta.ca*
- *Michail Alvanos is also with IBM CAS Reasearch, Toronto, Canada.*

accesses when occurring inside work-sharing constructs such as *upc_forall* in UPC. However, passing the number of processing nodes as a compiler flag is not usually a practical solution because it requires different binaries for different number of processors. Moreover, a number of available UPC benchmarks are not using the *upc_forall* loop structure.

One example of large-scale parallel machines is the IBM® Power® 775 supercomputer [15] used for performance evaluation in this paper. This is a distributed-memory machine that features as many as 16384 32-Core compute nodes, connected using a two-level direct-connect interconnect topology that fully connects every element in each of the two level through a Hub chip [16]. Designed for 0.948 Teraflop/s peak performance per node, the machine is IBM's answer for the DARPA's High Productivity Computing Systems (HPCS) initiative.

This paper presents an optimization to reduce the impact of fine-grained accesses, through a combination of compile-time (static) and runtime (dynamic) coalescing techniques. The goal is to improve application performance without hindering its programmability. The dynamic coalescing, based on a modification of the inspector-executor technique [17], [18], [19], [20], is used to discover the affinity between accesses and data allocation in the absence of explicit compile-time affinity information, and thus to enable the runtime to coalesce fine-grained accessed. Contributions of this work include:

- A combination of static and dynamic coalescing techniques to increase communication efficiency, tolerate network latencies, and decrease runtime overhead. A demonstration that runtime and static coalescing can improve the performance of fine-grained accesses inside loops.
- A thorough quantitative performance study with a comparison of the proposed code transformations with a manually optimized versions of the benchmarks. The experimental evaluation indicates that the proposed tranformations can achieve from 15% up to 63% of the performance of the hand-tunned versions.
- An evaluation of the scalability of the UPC language using benchmarks with fine-grained communication using the PERCS architecture. The interconnection network limits the performance for certain data access patterns.

2 Background

PGAS programming languages use the same programming model for local, shared and distributed memory hardware. The programmer sees a single, coherent, shared address space, where shared variables may be directly read and written by any thread.

The Unified Parallel C (UPC) language follows the PGAS programming model. It is an extension of the C programming language [21] designed for high-performance computing on large-scale parallel machines. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per core. The UPC language can be mapped to either distributed-memory machines, shared-memory machines or hybrid: clusters of shared memory machines.

```
int OUT[N][N];
int IN[N][N];

void stencil_kernel(){
  int i,j;
  for(i=1; i<N-1; i++ ){
    for(j=1; j<N-1; j++){
      OUT[i][j] = 0.25f * (IN[i-1][j] + IN[i+1][j]
                          + IN[i][j-1] + IN[i][j+1] );
    }
  }
}
```

Listing 1. Serial version of a stencil kernel.

Listing 1 presents a serial version of a stencil benchmark. A straightforward UPC parallel version is shown in Listing 2. Arrays `IN` and `OUT` are declared as `shared` (line 2), and their elements will be distributed cyclically among the threads. The construct `upc_forall` distributes loop iterations among the UPC threads. The affinity expression (`&OUT[i]`) in the `upc_forall` construct specifies that the owner thread of the specified element `&OUT[i]` will execute the ith loop iteration.

```
1  shared int OUT[N][N];
2  shared int IN[N][N];
3
4  void stencil_kernel(){
5    int i,j;
6    upc_forall(i=1; i<N-1; i++; &OUT[i] ){
7      for(j=1; j<N-1; j++ ){
8        OUT[i][j] = 0.25f * (IN[i-1][j] + IN[i+1][j]
9                            + IN[i][j-1] + IN[i][j+1] );
10     }
11   }
12 }
```

Listing 2. Parallel version of a stencil kernel.

In this example access locality to array `OUT` is exploited through the use of the affinity expression. UPC compilers typically translate each access to shared data to a runtime call to fetch or store data, leading to fine-grain communication, which may yield poor performance. Communication traffic in this case is $(N \times N \times 4)$ elements, with one access per element.

```
#define B N*(N/THREADS)
shared [B] int OUT[N][N];
shared [B] int IN[N][N];

void stencil_kernel(){
  int i,j;
  upc_forall(i=1; i<N-1; i++; &OUT[i] ){
    for(j=1; j<N-1; j++ ){
      OUT[i][j] = 0.25f * (IN[i-1][j] + IN[i+1][j]
                          + IN[i][j-1] + IN[i][j+1] );
    }
  }
}
```

Listing 3. Blocked parallel version of a stencil kernel.

A better distribution of the shared data can be achieved through the use of layout modifiers. Listing 3 shows a distribution by rows, where each thread owns n consecutive rows. Where $n = N/THREADS$ as specified by the `[B]` blocking factor. In this case, non-local read memory accesses are reduced to $(2 \times N \times THREADS)$ because only computation of positions in the boundary of each thread access remote data. But still one access per element is performed leading to fine-grained communication.

Overheads of fine grain accesses

When the physical data mapping is unknown at compile time and the programmer does not use the `upc_forall` loop structure, the compiler does not apply shared data coalescing or privatization optimizations. The compiler privatizes, or coalesces, shared data when there is affine information. Such information may be hard for the compiler to extract from the code without programmer assistance. Two problems arise from codes with fine-grained accesses to shared data: (i) low communication efficiency because of the use of small messages, and (ii) high overhead due to a large number of runtime calls created.

3 Related Work

Optimizations for data coalescing using static analysis exist in Unified Parallel C [12], [14] and High Performance Fortran e.g. [13], [22], [23]. A compiler uses data and control-flow analysis to identify shared accesses to specific threads and creates a single runtime call to access the data from the same thread.

However, in the UPC language, most of the solutions proposed require that the programmer specifies the number of threads, the number of processing nodes, and the blocking factor at compile time. Therefore, a different binary file is needed for each thread combination. Also, in practice, UPC applications do not make extensive use of the parallel loop construct, which means that a substantial number of accesses are left unoptimized. The *upc_forall* loop structure provides the compiler the necessary information about which iterations will be executed from a thread and what data each thread is going to access. In contrast, we provide a generic approach for coalescing data accesses at runtime without knowledge of physical data mapping and without the necessity to use the parallel loop structure.

Chavarria-Miranda and Mellor-Crummey propose static coalescing with symbolic (dynamic) number of processes [13]. However, their approach requires the usage of the ON HOME directive, which implies affinity information to compute where the data belong. This approach is similar to the usage of the affinity expression in parallel loop structure in UPC.

Chen and Yellick use a similar approach [12]. In both cases the runtime aggregates and double buffers the data. There are three main differences: (a) First, their approach does not work well with loops. Thus, the amount of data aggregation is limited. In contrast, the solution described in this paper focuses on loops that contain fine-grained communication and achieves much better aggregation and overlapping of communication and computation. (b) Second, they have the dilemma of using pipeline versus aggregation. The new solution proposed here uses both because the runtime analyzes many more shared access. (c) Third, they have to block to analyze the addresses, to decide what approach is better, and to issue accesses. In contrast, the solution here always tries to overlap computation and communication.

Another approach to minimize the communication latency in the PGAS programming model is to split the issuing of shared accesses and the synchronization points. This approach is called either “split-phase communication” [24] or “scheduling” [22], [25], [26]. However, these approaches have limited opportunities within a loop structure because of the complexity of data flow analysis.

The inspector-executor strategy is a well-known optimization technique for global name space programs for distributed execution and it has been used [18] for global-address-space language, or language-targeted optimizations such as High Performance Fortran [19], [23], Titanium language [20], X10 [27], and Chapel [28]. The typical solution is for the inspector loop to analyze the communication pattern and for the executor loop to perform the actual communication based on the results of the analysis performed in the inspector loop. The new approach described in this paper applies a strip-mining transformation on the original loop to achieve better overlapping of communication with computation in the loop.

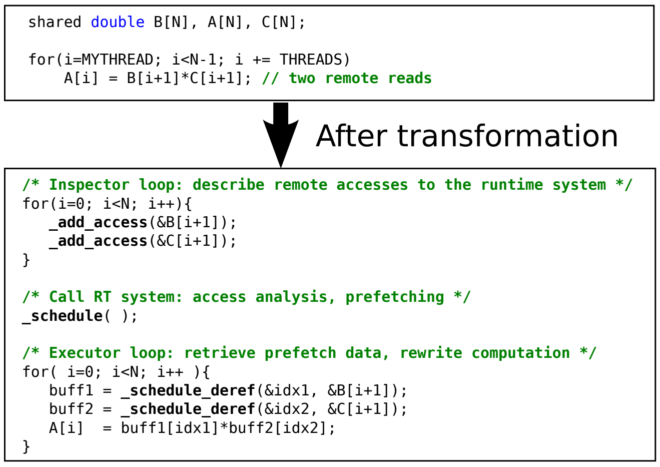
4 Design

This section presents a new method for run-time and compile-time coalescing of fine-grain accesses when the number of threads is unknown at compile time.

4.1 Runtime data aggregation

The goal is to identify remote fine-grain shared accesses and coalesce them together so that for each remote thread one coarser-grain communication transfers all the necessary data. Remote shared accesses will be collected, analyzed, coalesced, and fetched, ahead of time before the data is required. This process will happen at runtime when the physical data mapping is known.

This new method is based on the inspector-executor technique [17], [18], [19], which was re-architected for better performance and scalability. The inspector loop collects the shared addresses, then the runtime analyzes and aggregates them, and the executor loop



```
shared double B[N], A[N], C[N];

for(i=MYTHREAD; i<N-1; i += THREADS)
    A[i] = B[i+1]*C[i+1]; // two remote reads
```

```
/* Inspector loop: describe remote accesses to the runtime system */
for(i=0; i<N; i++){
   _add_access(&B[i+1]);
   _add_access(&C[i+1]);
}

/* Call RT system: access analysis, prefetching */
_schedule( );

/* Executor loop: retrieve prefetch data, rewrite computation */
for( i=0; i<N; i++ ){
   buff1 = _schedule_deref(&idx1, &B[i+1]);
   buff2 = _schedule_deref(&idx2, &C[i+1]);
   A[i]  = buff1[idx1]*buff2[idx2];
}
```

Fig. 1. The inspector-executor optimization.

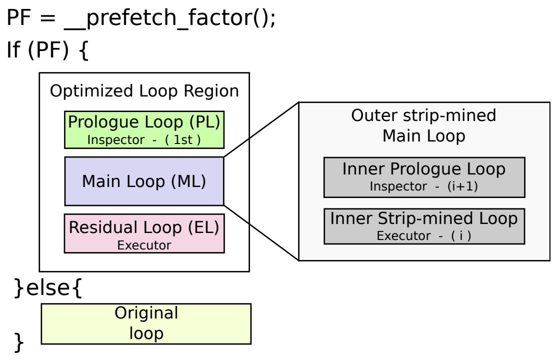


Fig. 2. The final form of transformed loop.

reads the data from local buffers to perform the actual computation. Figure 1 presents the transformation in the UPC language. However, the generic inspector-executor approach has two problems that this work addresses: (i) the *pause issue*: the execution of the actual program is paused to analyze shared accesses and to fetch corresponding data; (ii) the *resource issue*: the number of iterations of the loop may be so high that the memory requirements are increased to unacceptable levels.

To cope with these problems, the compiler strip-mines the main loop breaking the loop's iteration space into smaller chunks of size *prefetch factor* (*PF*). The value of *PF* is chosen by the runtime to maximize benefit without exhausting the resources, thus solving the *resource issue*. To address the *pause issue*, the compiler skews the inspector loops by one block, creating a pipelining effect. The inspector loop collects the elements for the $(i+1)^{\text{th}}$ block of iterations while the executor loop reads the coalesced data from a local buffer of the i^{th} block of iterations. The compiler applies loop versioning and creates two loop variants: the transformed and the native. The runtime performs a profitability analysis and decides which version to execute. Figure 2 presents the transformed loop structure.

In the final step of the transformation the necessary runtime calls are inserted for collecting (*__add_access*), aggregating (*schedule*), and recover from local buffers (*__dereference*) of shared accesses. Alvanos *et al.* described an earlier prototype and a preliminary evaluation of the runtime data aggregation in [29], [30].

4.2 Static coalescing

The goal of static coalescing is to coalesce shared accesses at compile time when the compiler can determine that the remote data belong to the same thread [31]. If the number of UPC threads is unknown at compile time, static coalescing is possible only when accessing members of shared structures that belong to the same thread. Therefore, the compiler applies the optimization when the program uses shared arrays with data structures.

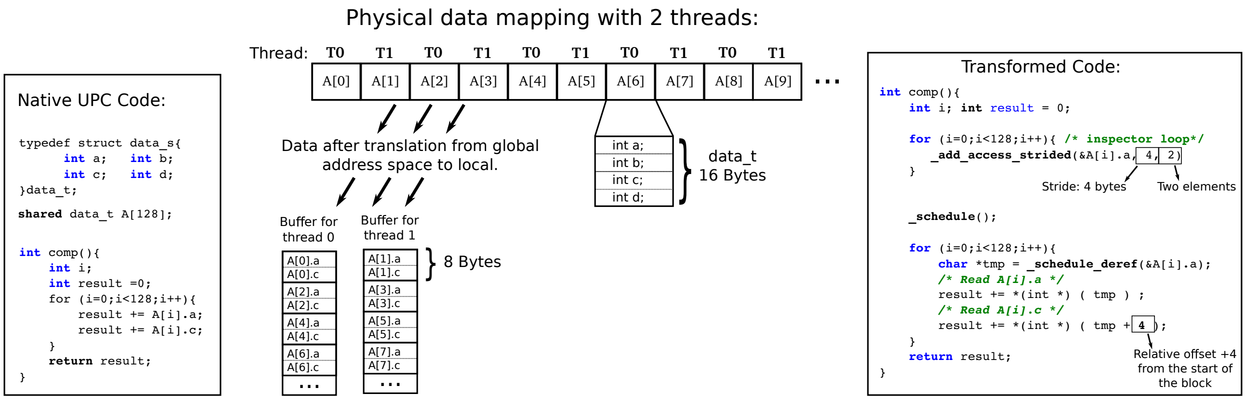
Figure 3 exemplifies the static-coalescing transformation. The example program in Figure 3(a) shows a simple reduction of the `a` and `c` struct fields, from a shared array of structures written in UPC. Figure 3(b) presents the physical mapping of the shared array running with two UPC threads. The array is distributed cyclically among the UPC threads. Figure 3(c) presents the final code transformation. The transformation identifies that accesses to `a` and `c` struct fields come from the same thread. Thus, it generates the appropriate runtime call (*__add_access_strided*) in the inspector loop to pass along the information about the stride between these accesses and the number of elements to fetch. At runtime, the coalescing optimization fetches the `a` and `c` fields and places them in consecutive memory locations in the local buffer.

5 Implementation

The new code transformations are prototyped in the XLUPC compiler framework [32] that uses the IBM PGAS runtime [33]. The compiler contains additional optimizations for UPC and other languages, including C and C++. The compiler applies the loop transformations and inserts the calls between different parts of the loop structure. The runtime is responsible for the profitability analysis, keeping the list of shared accesses, message aggregation, and retrieving the data from local buffers.

5.1 The Static coalescing algorithm

The implementation of the algorithm requires additional compiler analysis between shared references and runtime modifications. Algorithm 1 outlines the compiler's static analysis. First, the compiler analyzes the shared accesses that are fields of shared structures. The analysis classifies the shared addresses into buckets containing compatible shared addresses (line 6). A shared reference is compatible with a bucket when


Physical data mapping with 2 threads:
Thread: T0 T1 T0 T1 T0 T1 T0 T1 T0 T1
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9] ...
Data after translation from global address space to local.
int a; int b; int c; int d;
data_t 16 Bytes
Buffer for thread 0
Buffer for thread 1
A[0].a A[0].c A[2].a A[2].c A[4].a A[4].c A[6].a A[6].c ...
A[1].a A[1].c A[3].a A[3].c A[5].a A[5].c A[7].a A[7].c ...
8 Bytes
Stride: 4 bytes
Two elements
Relative offset +4 from the start of the block
(a)
(b)
(c)

Native UPC Code:

```
typedef struct data_s{
      int a;   int b;
      int c;   int d;
}data_t;
shared data_t A[128];

int comp(){
    int i;
    int result =0;
    for (i=0;i<128;i++){
        result += A[i].a;
        result += A[i].c;
    }
    return result;
}
```

Transformed Code:

```
int comp(){
    int i; int result = 0;

    for (i=0;i<128;i++){ /* inspector loop*/
       _add_access_strided(&A[i].a, 4, 2)
    }

    _schedule();

    for (i=0;i<128;i++){
        char *tmp = _schedule_deref(&A[i].a);
        /* Read A[i].a */
        result += *(int *) ( tmp ) ;
        /* Read A[i].c */
        result += *(int *) ( tmp + 4 );
    }
    return result;
}
```

Fig. 3. Example of Static data coalescing: native UPC source code (left), physical data mapping (middle), and transformed code (right). Transformed code is simplified for illustrative purposes.

```
AnalyseSharedRefs($Procedure\quad p$)

1:  $RefList \leftarrow collectSharedReferences()$;
2:  $BucketRefList \leftarrow \emptyset$;
3:  for each shared mem ref $R_s$ in $RefList$ do
4:      $isInserted \leftarrow FALSE$;
5:      for each shared Bucket $bck_s$ in $BucketRefList$ do
6:          if $R_s$ is compatible with $Bck_s$ then
7:              $Bck_s$.Add($R_s$);
8:              $isInserted \leftarrow TRUE$;
9:              break;
10:         end if
11:     end for
12:     if $isInserted$ = $FALSE$ then
13:         $Bck_s \leftarrow newShrReferenceBucket()$;
14:         $Bck_s$.Add($R_s$);
15:         $BucketRefList$.Add($Bck_s$)
16:     end if
17: end for
```

Algorithm 1: Analysis of shared references.

```
InsertSchedulerDereferenceCalls($Procedure\quad p$)

1:  for each Shared Bucket $bck_s$ in $BucketRefList$ do
2:      $R_{base} \leftarrow bck_s.getBaseSharedReference()$;
3:      $stmt \leftarrow innerloop_i.findLocation(R_{base})$;
4:      $innerloop_i^{stmt}.Add($ buffer = _sched_deref, &index);
5:      for each shared mem ref $R_s$ in $bck_s$ do
6:          $stmt_s \leftarrow SHARED\_STATEMENT(\ R_S\ )$;
7:          for each statement $stmt$ in $innerloop$ do
8:              if $stmt = stmt_s$ then
9:                  $innerloop.Replace(stmt^{expr}, buffer[index])$;
10:             end if
11:         end for
12:         $index \leftarrow index + 1$;
13:     end for
14:     $stmt \leftarrow innerloop_i.findLocation(R_{base})$;
15:     $innerloop_i^{stmt}.Add($ buffer = _sched_deref );
16:     for each shared mem ref $R_s$ in $bck_s$ do
17:         $stmt_s \leftarrow SHARED\_STATEMENT(\ R_S\ )$;
18:         for each statement $stmt$ in $epilogloop$ do
19:             if $stmt = stmt_s$ then
20:                 $epilogloop.Replace(stmt^{expr}, buffer[index])$
21:             end if
22:         end for
23:         $index \leftarrow index + 1$;
24:     end for
25: end for
```

Algorithm 2: Insertion of dereference calls.

the containing shared references use the same base symbol (array), the same array index, the same element access size, but different offset into the structure. If there are no compatible buckets, a new bucket is created for the shared reference (line 14). Finally, the analysis sorts the shared references during the addition to the bucket, based on the local offset. For each bucket the compiler inserts the dereference call on the first occurrence of a shared reference of the bucket and replaces each shared reference with the local buffer, by increasing the index of the local buffer based on the order of shared references.

Algorithm 2 presents the insertion of *_sched_deref* calls. For each bucket, the compiler inserts the dereference call on the first occurrence of a shared reference of the bucket and replaces each shared reference with the local buffer, by increasing the index of the local buffer based on the order of shared references.

5.2 Local data-access transformation

One of the key code transformations performed by the runtime system is to create an efficient access to shared data that belong to the same UPC thread. The runtime system identifies and ignores local shared accesses, thus avoiding the overhead of unnecessary analysis. The runtime system returns a pointer to the local data in the dereference calls to avoid memory copies. However, the static coalescing code transformation described in Section 5.1 requires symmetrical physical data mapping between the buffers. Thus, a constraint of that transformation is violated by accesses to local shared data because such accesses

Final Transformed Code:

```
int comp(){
    int i; int result =0 ; size_t stride;
    for (i=0; i<128; i++){
        stride = 4;   /* Compiler sets default stride */
        char *tmp = _schedule_deref(&stride, &A[i].a);

        result += *(int *) ( tmp ) ;  /* Read A[i].a */
        /* Read A[i].c  */
        result += *(int *) ( tmp + 1*stride );
    }
    return result;
}
```

Relative offset from the start of the block

Runtime Dereference Call

```
void *_schedule_deref(struct fat_ptr *ptr, int *stride){

    if ( ptr->node == CURRENT_NODE){
       /* No need to change the stride */
       return __get_local_data(ptr);
    }

    /* Change the distance of elements */
    *stride = ptr->elem_size;
    return __get_prefetched_data(ptr);
}
```

Fig. 4. Final code modification and a high level implementation of the runtime.

have different memory mapping, in contrast with the prefetched buffers that contain only the prefetched elements.

However, the actual location of data is known at runtime. Therefore, this problem is solved by modifying the *_sched_deref* runtime call to return the stride between the accesses. Figure 4 presents the generated code and a part of the dereference call in the runtime. In this example the distance between the fields is four bytes. The runtime sets the stride when the data are prefetched and stored in local buffer. On the other hand, the runtime returns a pointer to local data and does not change the default stride between elements (eight in the example), when the shared data are local. The compiler generates code for accessing different fields of the structure by multiplying the relative offset based on the order of shared references.

5.3 Runtime support

In providing functionality for these new transformations, the runtime system: (a) decides if the transformation is profitable, (b) stores information for the shared references, (c) analyzes the shared references and tries to coalesced them, and finally (d) retrieves the data from the local buffers.

The first task is to decide whether the loop transformation is beneficial and to calculate a prefetch factor (Algorithm 3). If there is a single PGAS node (line 1) the runtime executes the unmodified version of the loop. The unoptimized loop does not use network communication: the runtime uses simple loads and stores (memcpy) to transfer the data. The overhead of using thread communication is lower than the overhead of the scheduling optimization, which requires keeping information about shared accesses and analyzing them. The algorithm sets an upper limit to the number of iterations that can be prefetched to avoid overconsumption of memory resources (line 5). The runtime calculates the prefetch factor by dividing the upper limit by the number of shared elements. The prefetch factor must be the minimum between the upper bound minus one, and the calculated prefetch factor. Using the upper bound minus one ensures that the program will enter at least one time in the main loop. If the prefetch factor is less than two, then the original version is used.

The second task of the runtime is to collect and store information about shared accesses in the inspector loops. For each shared access the runtime stores information about the shared variable, the offset, the blocking factor (BF), element size (ES) and the remote thread. For each pair of variables and UPC thread the runtime inserts an entry on a hash table. On each entry of this hash table the runtime maintains an array of the offsets. The coalescing algorithm requires an additional library call for the inspector loops to support the collection of the shared references. The new library call has two additional arguments: the stride and the number of elements.

The third task is to analyze, coalesce and prefetch shared accesses. The runtime first sorts the collected offsets using the quicksort algorithm, removes duplicates, and prepares the vector of offsets to fetch. There are two reasons for sorting and removing duplicates from the offset list. First, the sorting makes the translation from shared index to local index for the buffer in the executor loops faster because a binary search is used. Second, removing duplicates decreases the transfer size in applications that have duplicates, such as stencil computation. Finally, the runtime prepares a vector of offsets to fetch data.

The final task is the data retrieval from the local buffers. The runtime returns a pointer to the local buffer and sets the proper index value. Internally the runtime first tries to calculate the index value

```
Prefetch_factor(int num_elems, int upper_bound)

 1: if $XLPGAS\_NODES == 1$ then
 2:     return 0;
 3: end if
 4: $max\_loop \leftarrow upper\_bound - 1$;
 5: $PF \leftarrow MAX\_FETCH/num\_elems$;
 6: if $PF \geq max\_loop$ then
 7:     $PF \leftarrow max\_loop$;
 8: end if
 9: if $PF \leq 2$ then
10:     $PF \leftarrow 0$;
11: end if
12: return PF;
```

Algorithm 3: Calculation of Prefetch Factor.

directly by using an auxiliary table as a hint. When the runtime fails to find directly the proper value, it searches for the index value in the offset table by using a binary search algorithm.

5.4 Resolving Data Dependencies

```
#define N 8192
int compute( shared int *ptr1,
             shared int *ptr2 ){
  int i;
  for(i=0;i<N-1;i++){
    ptr1[i] = ptr2[i];
  }
}
```

Listing 4. Example of possible pointer aliasing.

In a parallel loop with references and assignments using shared pointers the compiler may not be able to figure out the data dependencies and it would assume alias dependencies between the shared pointers. An example of this case is a loop that has references and assignments using shared pointers and without any information about the shared arrays. In this case, the compiler cannot determine the dependencies and assumes that there are dependencies between the shared pointers. Listing 4 presents an example of possible alias dependencies: the compiler is unable to determine if there is an overlap between the `ptr1` and `ptr2` pointers.

To keep memory consistent, the shared write call in the runtime has been modified to include an additional argument flag that notifies the runtime to make additional checks for outstanding transfers. The compiler sets this flag to true if there is an overlapping between shared addresses or if the compiler fails to detect the alias dependencies. When handling stores to shared data, the runtime has to consider three different cases:

- There is no overlapping between stores and prefetched shared data. In this case, the runtime does not execute any additional code.
- The runtime has prefetched the shared data to the local buffer and there is shared store that targets the prefetched data. The runtime issues the store of the remote shared data and updates the local buffer to maintain the consistency.
- There are stores on shared data that the runtime transfer to the local buffer. In this case the runtime waits for the transfer to complete and then overwrites the prefetched data.

Figure 5 presents the user's code and the runtime implementation. The compiler sets the variable *update_local* if it fails to determine the dependencies between variables.

6 Experimental Evaluation

This experimental evaluation assesses the effectiveness of the runtime data aggregation and static coalescing optimizations. Five code-generation strategies are compared using micro benchmarks and actual applications running on a computer with over 32000 cores.

6.1 Comparing Code-Generation Strategies

The evaluation compares five different code-generation strategies. Each strategy leads to a different executable binary for each benchmark. Some strategies are not applicable for all the benchmarks.

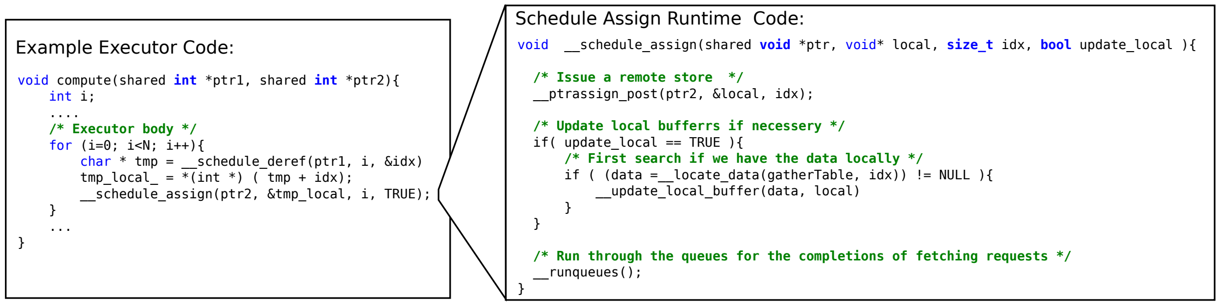
- *Baseline*: compiled with a dynamic number of threads and the inspector-executor optimization disabled. In the absence of information about the physical mapping of data, code transformation is limited to privatization of some shared accesses inside *upc_forall* loops.
- *Aggregation:* the compiler applies the runtime data-aggregation code transformation that prefetches and coalesces shared references at runtime (presented in section 4.1).
- *Aggregation + Coalescing*: perform static coalescing (presented in section 4.2) in addition to the runtime data aggregation.
- *Hand-optimized*: loops are manually strip mined to use coarse-grained communication and manual pointer privatization. This version also uses collective communication mechanisms whenever possible.
- *MPI*: uses coarse-grained communication and collective communication whenever possible.

6.2 Experimental Platform

This evaluation uses the IBM® Power® 775 supercomputer [15] to evaluate the optimization. It uses 1024 nodes with 32 POWER7® [34] cores on each node, running at 3.856 GHz, totalling 32768 cores. The POWER7 processor has 32-KByte instruction and 32-KByte L1 data cache per core, 256-KByte second-level cache per core, and a 32-MByte third-level cache shared per chip. Each core is equipped with four SMT threads and 12 execution units. The size of the available main memory is 128 GBytes. The machines are grouped in drawers consisting of eight nodes. Four drawers are connected to create a SuperNode (SN). The nodes are equipped with the POWER7 Hub chip interconnect [16] for communication. The Hub chip is connected with the four POWER7 chips using four links, of 24GB/s each. The Hub chip contains seven links for intra-drawer communication, 24 links for intra-SuperNode communication, and 16 links for inter-SuperNode communication.

6.3 Experimental Methodology

All runs use one process per UPC thread and schedule one UPC thread per POWER7 core. The upper limit for the number of iteration to prefetch (*MAX_FETCH*)


Example Executor Code:

```
void compute(shared int *ptr1, shared int *ptr2){
    int i;
    ....
    /* Executor body */
    for (i=0; i<N; i++){
        char * tmp = __schedule_deref(ptr1, i, &idx)
        tmp_local_ = *(int *) ( tmp + idx);
        __schedule_assign(ptr2, &tmp_local, i, TRUE);
    }
    ...
}
```

Schedule Assign Runtime Code:

```
void  __schedule_assign(shared void *ptr, void* local, size_t idx, bool update_local ){

  /* Issue a remote store  */
  __ptrassign_post(ptr2, &local, idx);

  /* Update local bufferrs if necessery */
  if( update_local == TRUE ){
      /* First search if we have the data locally */
      if ( (data =__locate_data(gatherTable, idx)) != NULL ){
          __update_local_buffer(data, local)
      }
  }

  /* Run through the queues for the completions of fetching requests */
  __runqueues();
}
```

Fig. 5. The runtime resolves dependencies with the help of the compiler.

is 4096. Each UPC thread communicates with other UPC threads by using the network interface or inter-process communication. UPC threads are grouped in blocks of 32 per node and one UPC thread is bound to each core. Each benchmark runs five times and the average of execution times is reported. Furthermore, we always execute one iteration of the optimized loop before the actual measurement, to warm-up the internal structures of runtime. In all cases the variation in execution time is less than 5%.

6.4 Benchmarks and Datasets

The optimizations described in this paper aim at benchmarks that contain fine-grained accesses.

Microbenchmarks: The microbenchmark is a loop that accesses a shared array of structures. There are four variations of the loop: (i) The loop accesses two consecutive array elements (stream like); (ii) The loop accesses two random elements of the array; (iii) The loop accesses four consecutive array elements (stream like); (iv) The loop accesses four random elements of the array. Listing 5 presents the (iii) variation.

```
typedef struct data{ double var0; double var1;
                     double var2; double var3;
} data_t;

#define SIZE (1<<31)
shared data_t Table[SIZE];

double bench_stream_4_fields(){
  uint64_t i;
  double result0 = 0.0, result1 = 0.0;

  for (i=MYTHREAD; i<SIZE-1; i+=THREADS){
   result0 = Table[i+1].var0 + Table[i+1].var1;
   result1 = Table[i+1].var2 + Table[i+1].var3;
  }
  return result0 + result1;
}
```

Listing 5. Microbenchmark kernel that reads four structure fields from a shared array.

Sobel: The Sobel benchmark computes an approximation of the gradient of the image intensity function, performing a nine-point stencil operation [35]. In the UPC version the image is represented as a two-dimensional shared array and the outer loop is a parallel *upc_forall* loop. A different data set size is used for each number of UPC threads (weak scaling), starting from 32768×32768 as input image size in 32 UPC threads, up to 1048576×1048576 using 32768 UPC threads. The maximum allocated memory is 2 TBytes in 32768 UPC threads.

Gravitational fish: The gravitational UPC fish benchmark emulates fish movements based on gravity. The benchmark is an N-Body gravity simulation using parallel ordinary differential equations [36]. There are three loops in the benchmark that access shared data, one for the computation of acceleration between fishes, one for data exchange, and one for the new position calculation. Different data set size is used for different number of UPC threads, starting from 16384 objects for 32 processors until to 524288 for 32768 processors.

WaTor: The benchmark simulates the evolution over time of predators and preys in an ocean [37]. The ocean is represented by a 2-D matrix where each cell can either be empty or contain an individual: a predator or a prey. In each time step predators and preys can move or replicate themselves, after a certain time period, to closer cells. Preys and predators can move or replicate only to empty cells while predators can eliminate a neighbor prey or die for starvation. The movement of preys occurs in a random way. Each UPC thread is assigned 32 lines of the ocean and the task size remains constant between different number of UPC threads.

Guppie: The guppie benchmark performs random read/modify/write accesses to a large distributed array. The selected size of data is static and evenly distributed among different UPC threads.

Mcop: The benchmark solves the matrix chain multiplication problem [38]. Matrix chain multiplication is an optimization problem where, given a set of matrices, the problem is to find the most efficient way to multiply these matrices. The matrix data is distributed along columns and communication occurs in the form of accesses to elements on the same row and column.

Other benchmarks containing coarse-grain accesses have also been evaluated showing neither improvement nor degradation in performance because the algorithm returns without any modification on the program if unable to find opportunities for coalescing

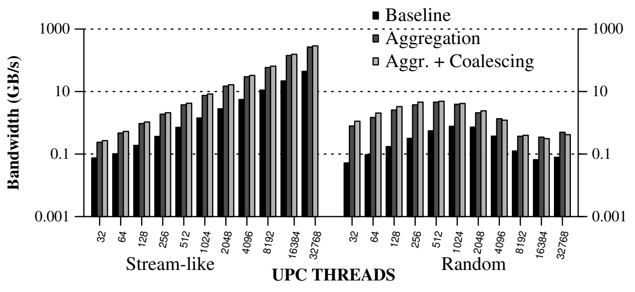


Fig. 6. Microbenchmark performance reading four fields from the same data structure in streaming fashion (left), and random reads using four fields (right).

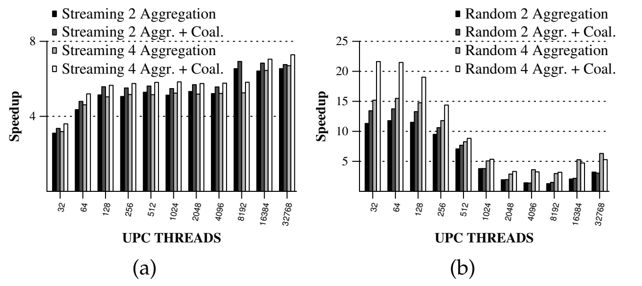


Fig. 8. Achieved speedup for the four microbenchmark variations reading in streaming fashion (left) and in random fashion (right).

and other optimizations are applied [39].

7 Experimental results

First the performance of the proposed transformations on microbenchmarks helps understand the maximum speedup that can be achieved and the potential performance bottlenecks. Then the performance of real applications compared with manual code improvements and to MPI versions provide a realistic view of the potential of the proposed transformations. benchmarks. Finally, an analysis of bottlenecks and limitations — along with the costs in terms of code increase, compilation time, and execution time — points to directions for future improvements.

7.1 Microbenchmark Performance

In the stream-like microbenchmark the bandwidth increases linearly with the number of UPC threads — notice the log-log scale in Figure 6. The stream-like microbenchmark reads data from the neighbouring UPC threads. The runtime is able to create one entry in a hash table resulting in very low memory overhead. On average, 4096 elements were coalesced into a single message. The speedup for this stream-like benchmark is between 3.1x and 6.7x as shown in the dark bars of Figure 7(a). The slight increase in the achieved speedup for more than 16384 UPC threads is most likely due to higher latency and network contention.

When reading elements in random order, the speedup varies from 3.2x up to 21.6x. The combination of the inspector-executor and the static coalescing optimizations gives a speedup of 10% over the simple inspector-executor approach in stream-like workloads and around from 10% up to 25% for the random one.

Comparing the achieved speedup between reading two elements in streaming fashion, the differences are negligible (Figure 8(a)). The performance is predictable and coalescing four elements statically is slightly better compared with coalescing only two elements, with the exception of 8192 UPC threads. However, the impact of static coalescing when using four elements is more profound compared with the coalescing of only two elements (Figure 8(b)). This effect is due to the bad performance of the baseline version, thus causing the amplification of performance gain of the optimized versions. Moreover the aggregation version has better results when coalescing four elements compared with the two-element version because more shared references are aggregated in runtime.

The random-access benchmark achieves better bandwidth than the stream-like variation when the prefetching is enabled and the benchmarks runs with 256 or less UPC threads. The Hub Chip architecture explains this result. The Hub Chip has seven different links for connecting nodes on the same drawer. These links have unidirectional bandwidth of 3 GB/s point-to-point between cores with a maximum of 24 GB/s aggregated unidirectional bandwidth [40]. In the streaming benchmark only one of the threads in the node communicates with a neighbouring node and therefore only one of the Hub links is used, decreasing the maximum bandwidth that is available. In contrast, in the random case, all nodes may communicate, with the communication potentially going through the seven available links.

The performance gain (speedup) of the random access decreases while it remains constant in the stream-like benchmark. There are two reasons for this behaviour: (i) The number of coalesced messages, and (ii) the memory consumption of the runtime. The right-side vertical axis of Figure 7(b) is the number of coalesced messages. As expected, there is a correlation between the speedup and the aggregation of the messages. In the stream-like benchmark all shared accesses come from the neighbour thread, therefore the number of coalesced messages remains constant and is determined by the ratio between the number of iterations inspected, *MAX_FETCH*, and the prefetch factor. In this micro benchmark 1024 messages were coalesced. Assuming a uniform distribution, the number of random accesses that hit the same thread decreases with the number of threads to the minimum of one array entry that contains 4 shared accesses for each element of the structure. Figure 7(c) presents

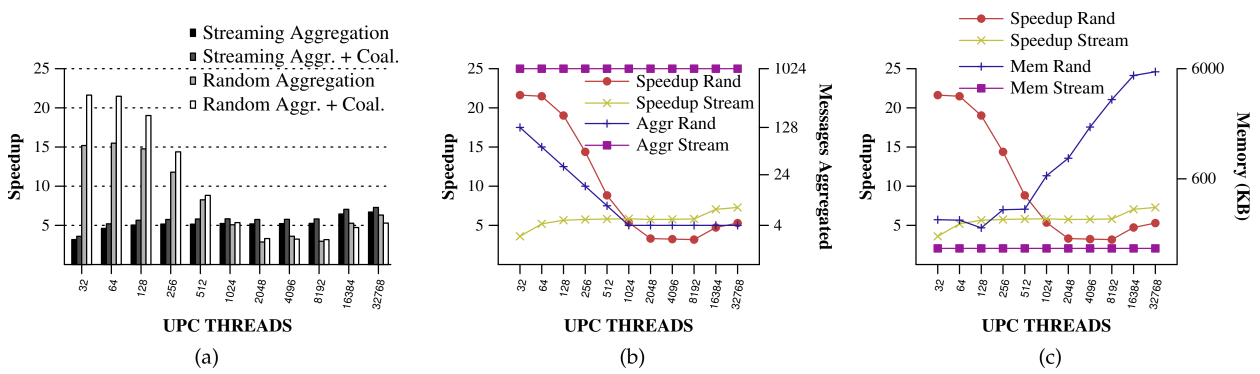


Fig. 7. Achieved speedup for the two microbenchmark variations reading four fields in steaming and random fashion (left), speedup compared with the number of messages aggregated (middle), and speedup compared with the memory consumption of the runtime (right).

the correlation between memory consumption and the speedup. In the stream-like benchmark memory consumption is kept constant because all accesses come from the same thread and only one entry of the hash table is required while a random distribution leads to a more populated hash table and therefore memory consumption increases with the number of threads.

The decrease in performance for random reads when there are more than 1024 UPC threads — with or without prefetch — is due to the interconnection between supernodes. The network architecture has a two-level direct-connect interconnect topology that fully connects every element in each of the two levels. With only two levels in the topology, the longest direct route L-D-L (intra - inter - intra supernode) [15] has at most three hops (two L hops and one D hop). This interconnection architecture limits the performance of random accesses pattern when most of the traffic is routed through remote links.

Assuming a uniform traffic distribution for the random-access-pattern benchmark, how much of the traffic uses the remote D links?

The experimental machine uses eight inter-supernode (8D) links to communicate [41]. Let S be the number of super nodes in the system. The number of links that each supernode has with other supernodes is $(S-1)\times 8$.

Each supernode has 32 nodes, and each node contains one Hub chip. Let N_R be the average number of remote links for each node:

$$N_R = \frac{(S-1)\times 8}{32} = \frac{(S-1)}{4}$$

The experimental evaluation uses 32 UPC threads per node. Let T_R be total number of UPC threads that use direct connection:

$$T_R = \frac{(S-1)\times 8\times 32}{32} = (S-1)\times 8$$

Let T_I be the total number of direct connections that a thread has inside the supernode. Let D be the number of direct connections that a node has with

UPC threads	1-hop Link (%)	Links > 1-hop (%)
1024	100 %	0 %
2048	50.39 %	49.61 %
4096	25.59 %	74.41 %
8192	13.18 %	86.82 %
16384	6.98 %	93.02 %
32768	3.88 %	96.12 %

TABLE 1
Percentage of traffic that uses remote and local links.

other nodes inside the supernode. Let I be the number of UPC threads in each node. All communication between these threads within a supernode are local. In this machine $D = 31$ and in this micro benchmark $I = 32$. Therefore $T_I = I \times D = 32 \times 31 = 992$

Let U be the number of UPC threads . The percentage of traffic that uses links with more than one hop is given by the following expression:

$$\frac{U-T_R-T_I-I}{U}$$

Table 1 presents the percentage of traffic that uses more than one hop. Using more than one hop, implies that the machine uses the slow remote links. Overall, the interconnection of the supernodes burdens the performance in random access patterns due to link saturation, with more than 1024 UPC threads.

7.2 Applications Performance

Figure 9(a) presents the performance results for the **Sobel benchmark** in mega-pixel per second. The runtime data *aggregation* optimization achieves a performance gain between +10% and +90%. The relatively low performance gain compared with the microbenchmark and the gravitational fish benchmark, is due to good shared data locality. For example, only 1.6% of the shared accesses are remote, running with 2048 UPC threads. The Sobel benchmark communicates with the neighbouring UPC threads only in the start and in the end of the computation. The *Aggregation*

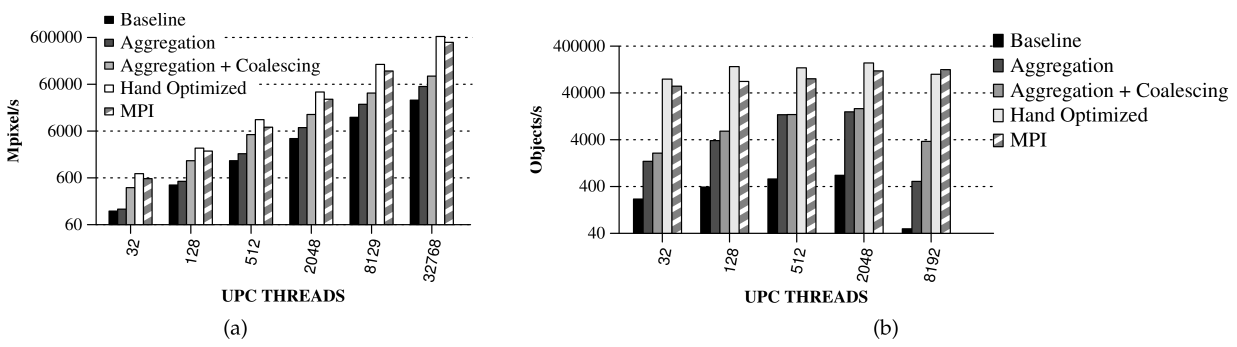


Fig. 9. Performance numbers for the Sobel benchmark (a), and fish benchmark (b) for different versions.

+ *Coalescing* technique achieves from 2.4x up to 3.3x speedup over the *baseline* because the static coalescing is able to decrease the number of library calls. Moreover, the Sobel benchmark achieves (*Aggregation + Coalescing*) from 27% up to 63% of the performance of the MPI version. Figure 11 shows the source of performance difference: the overhead from the inspector and executor loops. One interesting observation is that the UPC hand-optimized version is from 1.05x up to 1.45x faster than MPI version, because of the better overlap of one-sided communication.

Figure 9(b) reports the number of KBytes computed per second in the **fish benchmark**. The *static coalescing* (*Aggregation + Coalescing*) gives and additional speedup between 9x up to 26x compared with the baseline version. Furthermore, the performance drops significantly for more than 2048 UPC threads. The evaluation is limited to 8192 UPC threads, because runs with 16K or more threads are not practical. There are two issues that limit the performance of the application: (i) the architecture limitations of the interconnect network and (ii) the way the data are stored and accessed. First, for the same reasons that *random access microbenchmark* has bad performance for more than 1024 UPC threads, the fish benchmark saturates the inter-SuperNode links. Secondly, all UPC threads access data in streaming fashion starting from the first UPC thread. Thus, for the first iterations of the loop, all the UPC threads try to access at the same time the data on the first UPC thread. The compiler-optimized version (*Aggregation + Coalescing*) achieves from 8% up to 32% of the speed of the UPC hand-optimized version. The advantage of the UPC-optimized and MPI versions is the use of collective communication.

Figure 10(a) presents the performance numbers for the **WaTor benchmark** in KB/s. The *aggregation* gives a speedup from 3.8x up to 15.6x compared with the baseline version. Furthermore, the combination of *aggregation and static coalescing* is from 5.3x up to 25.1x faster than the baseline. The performance decreases for more than 1024 UPC threads because of the communication pattern. The benchmark reads 25 points of the neighbouring cells of the grid, in order to calculate the forces. The large number of remote shared references saturates the remote links for more than 2048 UPC threads. The compiler optimizes most of the remaining shared accesses using the remote update optimization [39]. The MPI version is faster but requires additional code before and after the force calculation and objects movement. The compiler does not create additional calls for accessing the data in contrast with the UPC versions.

Figure 10(b) presents the performance numbers for the **Guppie benchmark** in MegaUpdates/s. The *runtime aggregation* gives a speedup from 1.1x up to 7.5x compared with the baseline version. Furthermore, manual code modifications allows compiler to optimize using the remote update optimization [39]. The remote update optimization uses hardware acceleration. Thus, the achieved performance for the manual optimized code is from 1.1x up to 7.5x times faster than the automatically optimized version. The hardware solution provides even more efficient solution than the inspector-executor optimization. The combination of *aggregation and coalescing* does not have any impact to the application performance because the applications contains only scalar accesses and does not contain structs.

Figure 10(c) presents the performance numbers for the **MCop benchmark** in Operations/s. The *aggregation* gives a speedup from 3.2x up to 14.4x compared with the baseline version. Applying the code transformations and manual unroll the loops four times results the benchmark to run 1.1x up to 7.5x times faster than the baseline optimized version. The automatic compiler optimization is faster due to better overlapping of communication and computation.

7.3 Bottlenecks and limitations

The most relevant drawback of our optimizations is the overhead added by the inspector loops and the analysis of accesses at runtime. Figure 11 presents a breakdown of the normalized execution time, for the data-aggregation optimization and for the combination of the two optimizations (runtime aggregation plus static coalescing). The shared-pointer arithmetic

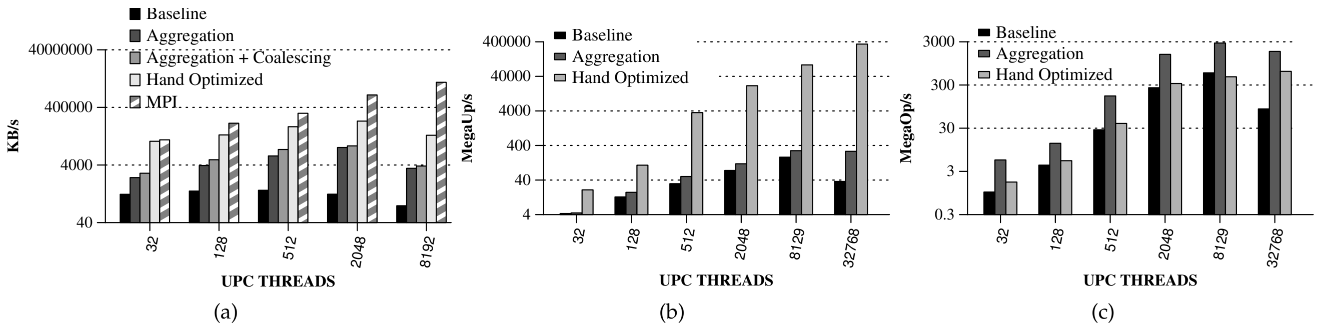


Fig. 10. Performance numbers for the WaTor (a), Guppie (b), and Mcop (c) benchmarks for different versions.

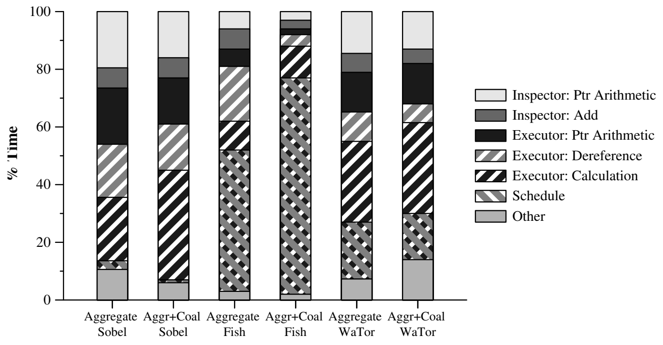


Fig. 11. Normalized execution time breakdown of the benchmarks using 128 UPC threads.

(*Ptr Arithmetic*) translates the offset to the relative offset inside the local prefetched data.

The inspector loops take 31% and 18% of the execution time in the Sobel and gravitational fish benchmarks, respectively. The static coalescing optimization decreases the overhead from the inspector and the executor loops from 20% up to 30%. The trend is similar for the gravitational fish and WaTor benchmarks. Finally, the fish benchmark is hindered by the poor performance of the scheduling algorithm (*Schedule*) which is due to the all-to-all communication pattern.

7.4 Cost of the optimization

The final part of the evaluation examines the transformation cost in terms of (a) compile time increase, (b) code-size increase, and (c) runtime memory increase.

The increase in compilation time varies from 20% to 35%. The main reason for this increase is the loop replication and the rebuilding of data and control flows for the transformed loops. The factor that affects the compilation time the most is the number of shared accesses. Compiler inserts runtime calls for inspecting the elements and to use the data from the local buffers.

The code-size increase provides an insight about the glue code that the compiler generates. The transformation requires the creation of three additional loops and the strip mining of the main loop. Moreover, it inserts some runtime calls to the end of inspecting and managing the shared accesses. Table 2 illustrates the code increase for the five benchmarks. The table also presents the number of calls created. The Sobel benchmark has the biggest increase in code, because it has a large number of shared accesses. The number of calls can be calculated using this equation:

$$\begin{aligned} number\_of\_calls &= number\_of\_accesses \times 4 \\ &\quad + loops\_optimized \times 6 \end{aligned} \tag{1}$$

For example, in the Sobel benchmark, the compiler creates eight calls per inspector loop and eight calls per executor loops. Moreover, the compiler creates two calls for the `__schedule`, two for the `__prefetch_reset`, one for `__prefetch_wait`, and one for `__prefetch_factor`. Thus, the total number of calls are $8 \times 4 + 1 \times 6 = 32 + 6 = 38$. On average, each prefetched shared access can add up to 2000 bytes of additional code. The cost per call is higher in the Guppie benchmark because the compiler prefetches only one element. On the other hand, MCop has the biggest increase because the compiler optimizes four different loops.

Finally, the optimization increases the memory requirements. The runtime keeps information about the shared accesses and uses local buffers to fetch data. Memory usage plays a key role in scaling to thousands of UPC threads thus its importance. To address this issue, the algorithm sets a maximum value for the prefetch factor which in turn limits the shared accesses that will be analyzed and therefore the memory footprint of the application. In the presented evaluation the prefetch factor limit was set to 2048, but it is configurable. In addition, the allocated memory for local buffers and metadata was limited to less than four MBytes to fit in the cache hierarchy.

8 Conclusions and Future Work

This paper presents an optimization to reduce the latency of fine grain shared accesses and to increase the overall performance of programs written in the UPC language. The optimization combines code transformations and run-time support to coalesce remote

Benchmark	Baseline	Aggregation	Aggregation & Coalescing	Diff	Num of Accesses	Cost per Access
Sobel	13702	17966	16270	+18.74 %	8	+856
Fish Grav	15619	19755	19179	+22.79 %	4	+890
WaTor	34872	37432	36792	+5.50 %	2	+960
Guppie	8533	11333	-	+32.81 %	1	+2800
Mcop	17851	17851	-	+69.64 %	8	+1554

TABLE 2
Object file increase in bytes. We consider only the transformed file.

accesses at compile time and at run time. This new optimization is evaluated using two microbenchmarks and five benchmarks to obtain scaling and absolute performance numbers on up to 32768 cores of a Power 775 machine. Our results show that the compiler transformation results in speedups from 1.15X up to 21X compared with the baseline versions and that they achieve up to 63% of the performance of the MPI versions.

Acknowledgments

George Almási and Ilie Gabriel Tanase supported the runtime development. Ioannis Manousakis, Ivan Tanasic, and Bo Wu provided valuable comments.

The researchers at Universitat Politècnica de Catalunya and Barcelona Supercomputing Center are supported by the IBM Centers for Advanced Studies Fellowship (CAS2012-069), the Spanish Ministry of Science and Innovation (TIN2007-60625, TIN2012-34557, and CSD2007-00050), the European Commission in the context of the HiPEAC3 Network of Excellence (FP7/ICT 287759), and the Generalitat de Catalunya (2009-SGR-980). IBM researchers are supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. The researchers at University of Alberta are also supported by the NSERC Collaborative Research and Development (CRD) program of Canada.

References

[1] J. Protic, M. Tomasevic, and V. Milutinovic, “Distributed shared memory: Concepts and systems,” *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 4, no. 2, pp. 63–71, 1996.

[2] B. Nitzberg and V. Lo, “Distributed shared memory: A survey of issues and algorithms,” *Computer*, vol. 24, no. 8, pp. 52–60, 1991.

[3] C. Amza, A. L. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, “TreadMarks: Shared Memory Computing on Networks of Workstations,” *IEEE Computer*, vol. 29, pp. 18–28, 1996.

[4] A. Itzkovitz and A. Schuster, “MultiView and Millipage – fine-grain sharing in page-based DSMs,” in *Proceedings of the third symposium on Operating systems design and implementation*, ser. OSDI ’99, 1999, pp. 215–228.

[5] U. Consortium, “UPC Specifications, v1.2,” Lawrence Berkeley National Lab Tech Report LBNL-59208, Tech. Rep.

[6] R. Numwich and J. Reid, “Co-array fortran for parallel programming,” Tech. Rep., 1998.

[7] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, “The Fortress Language Specification Version 1.0,” March 2008, http://labs.oracle.com/projects/plrg/Publications/fortress.1.0.pdf.

[8] Cray Inc, “Chapel Language Specification Version 0.8,” April 2011, http://chapel.cray.com/spec/spec-0.8.pdf.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” vol. 40, no. 10, pp. 519–538, Oct. 2005.

[10] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, “Titanium: A High-performance Java Dialect,” *Concurrency - Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[11] MPI Forum, “MPI: A Message-Passing Interface Standard.” http://www.mpi-forum.org.

[12] C. I. W. Chen and K. Yelick, “Communication optimizations for fine-grained upc applications,” in *In 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[13] D. Chavarria-Miranda and J. Mellor-Crummey, “Effective Communication Coalescing for Data-Parallel Applications,” in *In Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005, pp. 14–25.

[14] Christopher Barton, George Almasi, Montse Farreras, and Jose Nelson Amaral, “A Unified Parallel C compiler that implements automatic communication coalescing,” in *14th Workshop on Compilers for Parallel Computing*, 2009.

[15] R. Rajamony, L. Arimilli, and K. Gildea, “PERCS: The IBM POWER7-IH high-performance computing system,” *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 3–1, 2011.

[16] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, “The PERCS High-Performance Interconnect,” *High-Performance Interconnects, Symposium on*, vol. 0, pp. 75–82, 2010.

[17] J. H. Saltz, R. Mirchandaney, and K. Crowley, “Run-time parallelization and scheduling of loops,” *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, 1991.

[18] C. Koelbel and P. Mehrotra, “Compiling Global Name-Space Parallel Loops for Distributed Execution,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, 1991.

[19] P. Brezany, M. Gerndt, and V. Sipkova, “SVM Support in the Vienna Fortran Compilation System,” KFA Juelich, KFA-ZAM-IB-9401, Tech. Rep., 1994.

[20] J. Su and K. Yelick, “Automatic Support for Irregular Computations in a High-Level Language,” in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[21] ISO/IEC JTC1 SC22 WG14, “ISO/IEC 9899:TC2 Programming Languages - C,” Tech. Rep., May 2005.

[22] M. Gupta, E. Schonberg, and H. Srinivasan, “A unified framework for optimizing communication in data-parallel programs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 689–704, 1996.

[23] D. Yokota, S. Chiba, and K. Itano, “A New Optimization Technique for the Inspector-Executor Method,” in *International Conference on Parallel and Distributed Computing Systems*, 2002, pp. 706–711.

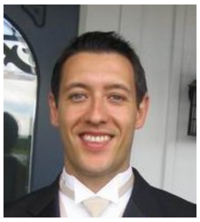
[24] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick, “Automatic nonblocking communication for partitioned global address space programs,” in *Proceedings of the 21st annual international conference on Supercomputing (ICS ’07)*, pp. 158–167.

[25] C. M. Barton, “Improving access to shared data in a partitioned global address space programming model. Ph.D. thesis,” 2009, University of Alberta.

[26] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey, "A Multi-Platform Co-Array Fortran Compiler," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04, pp. 29–40.
[27] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: Programming for hierarchical parallelism and non-uniform data access," in *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.
[28] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, and B. L. Chamberlain, "Global data re-allocation via communication aggregation in Chapel," in *SBAC-PAD*. IEEE Computer Society, 2012.
[29] M. Alvanos, M. Farreras, E. Tiotto, and X. Martorell, "Automatic Communication Coalescing for Irregular Computations in UPC Language," in *Conference of the Center for Advanced Studies*, ser. CASCON '12.
[30] M. Alvanos and E. Tiotto, "Data Prefetching and Coalescing for Partitioned Global Address Space Languages," Oct. 24 2012, US Patent App. 13/659,048.
[31] M. Alvanos, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell, " Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC," in *Proceedings of the 27th annual international conference on Supercomputing (ICS '13)*.
[32] G. Tanase, G. Almási, E. Tiotto, M. Alvanos, A. Ly, and B. Daltonn, "Performance Analysis of the IBM XL UPC on the PERCS Architecture," Tech. Rep., 2013, IBM RC25360.
[33] G. I. Tanase, G. Almási, H. Xue, and C. Archer, "Composable, Non-blocking Collective Operations on Power7 IH," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12, pp. 215–224.
[34] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, "Power7: IBM's Next-Generation Server Processor," *Micro, IEEE*, vol. 30, no. 2, pp. 7 –15, march-april 2010.
[35] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: a NPB experimental study," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02, Los Alamitos, CA, USA, 2002, pp. 1–26.
[36] S. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithms*, ser. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 2003.
[37] A. K. Dewdney, "Computer recreations sharks and fish wage an ecological war on the toroidal planet wa-tor," *Scientific American*, pp. 14–22, 1984.
[38] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education.
[39] C. Barton, C. Cascaval, G. Almasi, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral, "Shared memory programming for large scale machines," *Programming Language Design and Implementation (PLDI)*, pp. 108–117, June 2006.
[40] K. J. Barker, A. Hoisie, and D. J. Kerbyson, "An early performance analysis of POWER7-IH HPC systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, pp. 42:1–42:11.
[41] Redbooks, IBM, *IBM Power Systems 775 for AIX and Linux HPC Solution*, 2012.

Michail Alvanos Michail Alvanos has a PhD from the Universitat Politecnica de Catalunya. He was working as a resident student in the Programming Models research group at the Barcelona Supercomputing Center. He has been collaborating with the IBM Toronto Laboratory since 2010, working on static compilation optimizations for the XLUPC compiler. His research interests include parallel programming models and heterogeneous architectures. He received the MS degree in Computer Science from the University of Crete.

Montse Farreras Dr. Montse Farreras received her PhD degree in computer science at UPC (Universitat Politecnica de Catalunya) in 2008. She works as an associate professor at the same University, and she joined the Programming Models research line at BSC (Barcelona Supercomputing Center). In this research group she is conducting research about Parallel Programming Models for High Performance Computing (HPC), focusing on productivity, performance and scalability. She has been collaborating with the Programming Models and Tools for Scalable Systems group at IBM TJ Watson Research Institute since 2004, and contributed to the scalable Runtime System for the XLUPC compiler.



Ettore Tiotto Ettore Tiotto graduated "Summa Cum Laude" in Applied Physics from the University of Torino, Italy, in 1996. Ettore joined the IBM Toronto Laboratory in 1999, were he held several development positions within the static compilation technology team. His research interests include compilers, runtime systems, and programming languages. Since 2007 Ettore has participated in the ongoing research and development of a Unified Parallel C compiler (recipient of the High Performance Computing Challenge Class 2 Award). He currently leads the UPC compiler development team in the IBM Toronto Laboratory.

Jose' Nelson Amaral José Nelson Amaral is a professor of Computing Science at the University of Alberta, Canada. He received a Ph.D. in Electrical and Computer Engineering from the University of Texas at Austin, in 1994, an M.E. from the Instituto Tecnológico de Aeronautica, São José dos Campos, SP, Brazil, and the B.E. from the Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), RS, Brazil. His current research interests include Compiler Design, Static Analysis, Feedback Directed Compilation, Computer Architecture, High-Performance Computer Systems, Transactional Memory and the application of learning methods to the design of compilers. Amaral has published extensively and has served in many program committees for international conferences. Amaral is a Senior Member of the IEEE and of ACM and a Distinguished Speaker for ACM.

Xavier Martorell Xavier Martorell received the M.S. and Ph.D. degrees in Computer Science from Universitat Politecnica de Catalunya (UPC) in 1991 and 1999, respectively. He has been an associate professor in the Computer Architecture Department at UPC since 2001, teaching on operating systems. His research interests cover the areas of paralellism, runtime systems, compilers and applications for high-performance multiprocessor systems. Since 2005 he is the manager of the team working on Parallel Programming Models at the Barcelona Supercomputing Center. He has participated in several european projects dealing with parallel environments (Nanos, Intone, POP, SARC, ACOTES). He is currently participating in the European HiPEAC Network of Excellence, and the ENCORE, Montblanc and DEEP european projects.