DOI 10.1016/j.parco.2016.03.002

Using Shared-Data Localization to Reduce the Cost of Inspector-Execution in Unified-Parallel-C Programs

Michail Alvanos[a,c,1,*], Ettore Tiotto[2,b], José Nelson Amaral[3,e], Montse Farreras[5,d], Xavier Martorell[4,d]

[a] *Barcelona Supercomputing Center, C/ Jordi Girona, 1-3, Barcelona, Spain, 08034*
[b] *IBM Toronto Software Laboratory, Warden Ave, Markham, ON L6G 1C7, Canada*
[c] *IBM Canada CAS Research, Warden Ave, Markham, ON L6G 1C7, Canada*
[d] *Department of Computer Architecture, Universitat Politècnica de Catalunya, C/ Jordi Girona, 1-3, Barcelona 08034, Spain.*
[e] *Department of Computing Science, University of Alberta, Athabasca Hall (ATH) 342, T6G 2E8, Edmonton, Alberta, Canada*

Abstract

Programs written in the Unified Parallel C (UPC) language can access any location of the entire local and remote address space via read/write operations. However, UPC programs that contain fine-grained shared accesses can exhibit performance degradation. One solution is to use the inspector-executor technique to coalesce fine-grained shared accesses to larger remote access operations. A straightforward implementation of the inspector-executor transformation resultin excessive instrumentation that hinders performance.

This paper addresses this issue and introduces various techniques that aim at reducing the generated instrumentation code: a shared-data localization transformation based on Constant-Stride Linear Memory Descriptors (CSLMADs) [32], the inlining of data locality checks and the usage of an index vector to aggregate the data. Finally, the paper introduces a lightweight

*Corresponding author
[1]Email: malvanos@gmail.com
[2]Email: etiotto@ca.ibm.com
[3]Email: amaral@cs.ualberta.ca
[4]Email: xavim@ac.upc.edu
[5]Email: mfarrera@ac.upc.edu

loop code motion transformation to privatize shared scalars that were propagated through the loop body.

A performance evaluation, using up to 2048 cores of a POWER 775, explores the impact of each optimization and characterizes the overheads of UPC programs. It also shows that the presented optimizations increase performance of UPC programs up to 1.8× their UPC hand-optimized counterpart for applications with regular accesses and up to 6.3× for applications with irregular accesses.

1. Introduction

New parallel languages and programming models provide simpler means to develop applications that can run on parallel systems without sacrificing performance. Partitioned Global Address Space (PGAS) languages [16, 31, 2, 18, 12, 37] 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. Unified Parallel C (UPC) [16], an extension of the C programming language, follows the PGAS programming model.

PGAS languages offer the advantage of fast development of parallel applications, in comparison with the Message Passing Interface (MPI) [30], through the use of a shared-memory abstraction. These programs may contain fine-grained shared accesses that lead to performance degradation. Often, after initial coding, the programmer fine tunes the source code to produce a more scalable version. However, the reality is that, at the end of this tuning for performance, the PGAS code resembles its MPI equivalent. Therefore, the performance tuning often nullifies the ease-of-coding and ease-of-maintenance advantages of PGAS languages.

A solution to improve the performance of fine-grained communication is to apply compiler and runtime optimizations to reduce the cost of inter-node communication. Proposed methods to improve fine-grained communication in PGAS languages include inspector-executor transformation [27, 11, 5], static coalescing [14, 8, 25], limited privatization [10, 15], and software caching [39]. However, a big hurdle in the code generation of UPC language is that the compiler ends up inserting runtime calls to transform UPC "shared" accesses into requests for data (or actions) to other address partitions. Thus, an important question to answer is: *how can we achieve performance comparable to C or MPI using the UPC programming model?* Despite the great

work done both with High-Performance Fortran and UPC, today no compiler delivers acceptable performance in the case of fine-grained communication.

This paper focuses on solving the problem of excessive instrumentation produced from the compiler in UPC language. This paper extends previous work presented at the SBAC-PAD conference [3] by (i) including a detailed presentation of the shared-reference-aware loop-invariant code motion and privatization, (ii) better performance evaluation including comparison with the serial version of benchmarks and parameter exploration, and (iii) broader related work for completeness. The main contributions of this paper are:

- The combination of a shared-data localization transformation based on Constant-Stride Linear Memory Descriptors (CSLMADs) [24] and a new shared-reference-aware loop-invariant code motion and privatization — which is designed specifically for UPC language — to deliver improved performance for benchmarks that rely on fine-grained communication. This transformation improves the performance of programs containing fine-grain accesses by orders of magnitude.

- A thorough performance study of this combined approach using the IBM Power 775 architecture [33]. It also includes a parametrization and characterization of overheads of UPC programs. These analysis indicate that this approach is an important step toward delivering the promised combination of productivity and performance through the use of the UPC language.

The rest of this paper is organized as follows. Section 2 reviews the Unified Parallel C language and the challenges of fine-grained communication. Section 3 presents the optimizations for the inspector-executor approach. Section 4 describes the experimental methodology. The results of the evaluation appear in Section 5. Section 6 discusses previous work that is related to this research. We present the conclusions and the future work in section 7.

2. Unified Parallel C

The Unified Parallel C (UPC) language follows the PGAS programming model. It is an extension of the C programming language 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.

```
1  typedef struct fish { double x; double vx;
2                        double y; double vy; } fish_t;
3  typedef struct f_acc { double ax; double ay; } fish_accel_t;
4
5  shared [NFISH/THREADS] fish_t fish[NFISH];
6  shared [NFISH/THREADS] fish_accel_t acc[NFISH];
7
8  for each time step {
9     upc_forall (i=0; i<NFISH; ++i; &fish[i]) { /* Force calc */
10       tmpx = tmpy = 0;
11       for (j = 0; j < NFISH; ++j) {
12          dx = fish[j].x - fish[i].x;
13          dy = fish[j].y - fish[i].y;
14          a = calculate_force(dx,dy);
15          tmpx += a * dx / r;
16          tmpy += a * dy / r;
17       }
18       acc[i].ax = tmpx; acc[i].ay = tmpy;
19    }
20     ... /* max_norm calc & Fish movement */
21 }
```

Listing 1: UPC version of gravitational Fish.

Listing 1, presents the computation kernel of the fish gravitational benchmark. The benchmark emulates fish movements based on gravity. The benchmark is an N-Body gravity simulation that solves ordinary differential equations in parallel [1]. Arrays `fish` and `accel` are declared as shared (lines 5-6). Shared arrays and shared objects are accessible from all UPC threads. The layout qualifier `[NFISH/THREADS]` specifies that the shared object is distributed in blocked form to different UPC threads. The construct `upc_forall` (line 9) distributes loop iterations among the UPC threads. The fourth expression in the `upc_forall` construct is the affinity expression. The affinity expression specifies that the owner thread of the specified element executes the i^{th} loop iteration. The UPC compiler translates the shared accesses to runtime calls. Runtime calls are responsible for fetching, or modifying, the requested data implying fine-grained communication. Two problems arise from codes with fine-grain accesses to shared data: (i) inefficient communication because of the exchange of short messages, and (ii) high overhead because of the number of runtime calls created.

The compiler deals with short messages by applying UPC-specific optimizations [5, 14, 8, 13, 10, 15, 39]. However, the overhead caused by the large number of runtime calls hinders performance even when the application exhibits good data locality. For example, a sequential C version of the Fish benchmark using 16384 objects takes 155 seconds while the UPC version —

without the optimizations described in this paper — takes 727 seconds on a single UPC thread in an IBM® POWER7® machine.

In this example, the compiler privatizes the accesses `fish[i].x` and `fish[i].y` in lines 12 and 13. However, the accesses `fish[j].x` and `fish[j].y` are not privatized because the program accesses the full shared array. In this case, the prefetching optimization of the compiler [4] transforms the loop into an inspector-executor form and aggregates, at runtime, the shared accesses. Listing 2 presents a simplified version of prefetching using the inspector-executor loop transformation. There are three entry points: the `__sched_add_access`, the `__sched_dereference`, and the `__schedule` calls. Before accessing shared pointers, the compiler also creates calls to shared pointer arithmetic (`__ptr_arithmetic`). The shared pointer is a *fat pointer* that contains information about the offset and the thread [7].

```
1  ...
2     upc_forall (i=0; i<NFISH; ++i; &fish[i]) {
3        for (j=0; j<NFISH; j++ ){
4           ptr = __ptr_arithmetic(&fish[j].x);
5           __sched_add_access(ptr,...);
6           ptr = __ptr_arithmetic(&fish[j].y);
7           __sched_add_access(ptr,...);
8        }
9        __schedule(); /* Schedule shared accesses */
10       for (j = 0; j < NFISH; ++j) {
11          ptr1 = __ptr_arithmetic(&fish[j].x);
12          tmp1 = __sched_dereference(ptr1,...);
13          ptr2 = __ptr_arithmetic(&fish[j].y);
14          tmp2 = __sched_dereference(ptr2,...);
15          ...
16       }
17    }
18 ...
```

Listing 2: Simplified example of inspector-executor.

3. Inspector-executor improvements

The experimental prototype for the code transformations described in this paper is built on top of the XLUPC compiler framework [36].

The biggest drawback of the inspector-executor code transformation is the overhead of function calls that the compiler introduces in order to inspect which data transfers are amenable for coalescing. Therefore, an important goal is to decrease the overhead of inspector-executor transformations by reducing the number of function calls executed at run time. Figure 1 presents the algorithm used to optimize loops with fine-grain accesses using

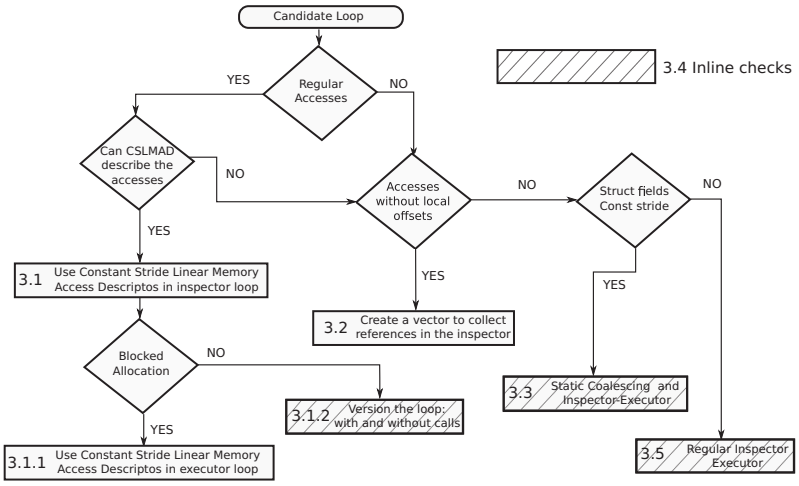


Figure 1: Compiler transformation algorithm.

the inspector-executor transformation. A hatched block in Figure 1 represents the cases where the inline transformation inserts branches between the entry points of the runtime, as explained in Section 3.4. After categorizing the access pattern into regular or irregular, the compiler analyses the stride to select an appropriate code transformation. A loop with regular accesses [32] is transformed using the CSLMAD framework (Case 3.1) and all the runtime calls are removed from the inspector loops (Case 3.1.1). Alternatively, for arrays that are not allocated in blocking fashion, two versions of the loop are created: one with run-time calls and the other without (Case 3.1.2). In the case of irregular accesses, the compiler creates a temporary array to collect the elements (Case 3.2). Finally, if the programmer uses aggregated data types (such as structs in C), the compiler tries to apply static coalescing (Case 3.3) or applies the original form of inspector-executor transformation (Case 3.5). Furthermore, the compiler inserts inline checks (case 3.4) in some for the cases to check for data locality and check for loop-invariant privatization possibilities.

3.1. Constant-Stride Linear Memory Descriptors

An array access analysis based on the Constant-Stride Linear Memory Access Descriptors (CSLMADs) identifies the type of access in a loop [24]. If the accesses on a shared array are regular, then the calls in the inspector loops are replaced with a single call.

CSLMADs are a restricted form of Linear Memory Descriptors [32] used to describe array accesses. CSLMADs lead to much simpler code transforma-

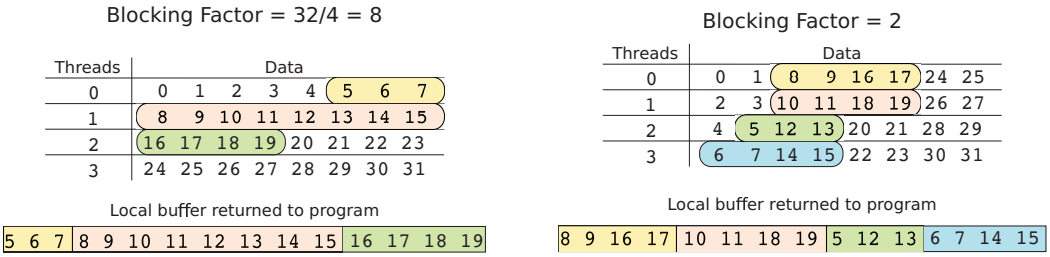


Figure 2: The shared address translation problem, when accessing the range 5-19.

tions than the more general Linear Memory Access Descriptors description. The main restriction that differentiates a CSLMAD from an LMAD is that a CSLMAD cannot represent overlapping indexing expressions. CSLMADs still capture a surprisingly large set of index expressions that appear in numerical applications — it also captures all the expressions that appear in the benchmarks used in this paper.

Each array access in CSLMAD form can be expressed as:$f(x) = b + a \times x$. The constant a is the stride of the CSLMAD and the integer constant b is the base of the CSLMAD (offset). Using the loop-range information the compiler transforms the descriptors to the following format: $\langle a, \mathit{local\_offset}, \mathit{low\_bound} + b, \mathit{upper\_bound} + b \rangle$. The constants *local_offset* are used to access fields of aggregated data types.

The transformation that replaces multiple calls in the inspector loop with a single call requires a normalized loop with monotonically increasing loop index. Furthermore, if shared arrays were allocated in blocked fashion, then the calls in the executor loops can be removed. In the UPC language the compiler detects blocked allocation in two cases: (i) when the programmer specifies the number of threads at compiler time and the blocking factor is the size of the array divided by the number of UPC threads; (ii) when the programmer uses a structure that contains an array. If the blocking factor is the size of the array divided by the number of threads, then the fetched data are placed in order. Figure 2 presents an example illustrating the fetching remote data with range from 5 to 19. On the left of the figure, the runtime fetches and places the data, in order, to the local buffer. In contrast, when the blocking factor is two, the runtime fetches the data but the placement is shuffled (right of the Figure 2).

3.1.1. Runtime improvements

The runtime uses a compact form to keep track of shared accesses if the shared array is allocated in blocked form. The runtime stores the shared accesses in the form of: ⟨*stride, local_offset, lower_bound, upper_bound*⟩. Hence, an additional benefit of using the compact representation form in the runtime is that the accesses do not require additional analysis. Furthermore, the runtime fetches the elements from the remote UPC threads in order. Also, the runtime tries to merge different CSLMADs when the descriptors have the same shared base array and stride. Thus, there is no duplication in the data transfers. Moreover, the runtime reuses the internal data structures for subsequent iterations by setting the new range of iterations to inspect.

3.1.2. CSLMADs in dynamic environments

Another challenge the compiler must address is the usage of shared pointers when the number of threads are not available at compile time. In this case, the compiler produces two variants of the executor loop. The first assumes that the loop has blocked allocation and the second assumes that the loop has blocking factor other than ideal. The compiler adds a branch to verify that all the arrays accessed are in blocked fashion. Thus, when any of the arrays has a blocking factor that is not ideal, the program executes the loop with the calls.

3.2. Usage of vectors to collect shared indexes

The compiler analyzes shared accesses occurring in irregular fashion to check if they access more complex structures, such as shared arrays of aggregated data types. For irregular accesses on shared arrays of native types, a temporary array (vector) stored in the stack is used to collect the shared indexes (Case 3.2 in Figure 1). The code generation inserts a call to inspect the elements at the end of each inspector loop. Internally the runtime processes the elements one by one. Thus, the performance gain is limited compared with the previous approach. The main benefit of this solution is a reduction in the number of calls in the inspector loop.

3.3. Combining dynamic with static Coalescing

Finally, the array-access analysis checks if the loop contains shared accesses to fields of aggregated data types that have constant stride. The compiler uses a previously proposed combination of dynamic and static coalescing methods [4]. The algorithm coalesces shared accesses when the compiler can

prove that the remote data belongs to the same thread (Case 3.1.1 in Figure 1). This is possible when accessing members of shared structures that belong to the same thread. Therefore, the compiler applies this optimization when the program uses shared arrays with data structures.

3.4. Inline checks

```
1   shared int A[128];
2   for (i=0;i<PF;i++){ // Inspector loop. shared ptr is fat pointer
3      ptr = __ptr_arithmetic(&A[i]);
4      if ( ptr.thread != MYTHREAD ) __sched_add_access( ptr, ...);
5   }
6   __schedule();
7   for (i=0;i<PF;i++){ // Similar approach in the executor loop:
8     ptr = __ptr_arithmetic( &A[i],... ):
9     if ( ptr.thread != MYTHREAD ){ local_ptr = __sched_dereference(ptr);
10    } else { local_ptr = CALC_LOCAL(ptr); } // Simple pointer additions
11    ... = local_ptr;
12  }
```

Listing 3: Example of code modifications for the inline checks.

A number of UPC applications contain shared references that target the local address partition but cannot be proven to be local at compile time. In applications that have good data locality only a small proportion of shared accesses are remote. For instance, in the Sobel benchmark [23] only 1.6% of the shared accesses are remote when running with 2048 UPC threads.

To solve this problem, this paper proposes the idea of inlining. The inline check optimization (Case 3.4 in Figure 1) inserts a branch before two entry points of the runtime: the *__sched_add_access* and the *__sched_dereference* calls. These branches check if the data accessed are remote or local. When the shared accesses are local the runtime avoids collecting the accesses. Instead, the executor loops use *thin pointers* to read the local data. The compiler applies this transformation, in addition to the transformations described earlier, when the benchmark exhibits irregular access pattern (Listing 3).

3.5. Shared-reference-aware loop-invariant code motion and privatization

Loop-invariant code motion is a traditional compiler optimization that moves statements and expressions that are not affected by the loop computations placing them outside of the loop body. It is, however, often difficult to prove that statements that use shared scalars and pointers are loop-invariant. Furthermore, copy propagation interferes with loop-invariant code motion because the existence of shared variables is often ignored.

There are two ways to solve this issue: (a) disable copy propagation; (b) implement an alternative code-invariant motion specific for PGAS languages. Disabling copy propagation can have negative effects in performance because this copy propagation may lead to dead code elimination. Thus, the implementation uses the second approach to implement a lightweight version of loop-invariant code motion in loops that have shared accesses.

```
UPCCodeInvariantMove(Procedure p)
 1: for each candidate loop structure L_i in p do
 2:    RefList ← ∅;
 Phase 1 - Gather Candidates
 3:    for all Shared reference R_s in L_i do
 4:       if R_s is loop-invariant then
 5:          RefList.Add(R_s);
 6:       end if
 7:    end for
 Phase 2 - Replace shared references in the loop
 8:    for each shared mem ref R_s in RefList do
 9:       stmt_s ← SHARED_STATEMENT( R_S )
10:       L_i^PROLOG.Add(tmp_var = R_s.SharedExpr)
11:       for each statement stmt in L_i do do
12:          if stmt = stmt_s then
13:             innerloop_i^stmt.Replace(stmt_s^expr, tmp_var)
14:          end if
15:       end for
16:    end for
17: end for
```

Algorithm 1: UPC loop-invariant code movement

Algorithm 1 presents this new loop-invariant code motion. For each shared reference, the algorithm uses the reaching-definition analysis using the Static Single Assignment (SSA) [19] representation. For each independent shared reference, the algorithm stores the shared value to a temporary scalar variable before the loop and replaces the occurrences inside the loop body.

4. Experimental Methodology

This evaluation uses an IBM® Power® 775 supercomputer [33] with 64 nodes with 32 Power7 [26] cores on each node, running at 3.856 GHz, totaling 2048 cores. The POWER7® (P7) processor has 32 KBytes instruction and

Benchmark	Description	Transformations	Communication Type
Stream-like	Microbenchmark: read data from the next thread.	Prefetch: Case 3.3	Stream-like from neighbour thread
Random-access	Microbenchmark: read data randomly.	Prefetch: Case 3.3	Random access
`Sobel` [23]	Gradient approximation of image intensity, using a 9-point stencil.	Prefetch: Case 3.1.1	Nine-point Stencil
`Fish Grav` [1]	N-Body gravity simulation using fishes as objects.	Prefetch: Cases 3.1.1/3.1.2	All-to-all/Reduction
`WaTor` [21]	Simulates the evolution over time of predators and preys in an ocean.	Prefetch: Cases 3.3, 3.4, and loop-invariant privatization	Random updates /Reduction/25-point stencil
`Mcop` [17]	Matrix chain multiplication problem: finds the most efficient way to multiply these matrices together.	Prefetch: Cases 3.1.1/3.1.2/3.2/3.4	All-to-all / Irregular
`Guppie` [29]	Random read/modify/write accesses to a large distributed array.	Prefetch: Cases 3.2/3.4	Random updates

Table 1: Benchmarks and Communication type.

32 KBytes L1 data cache per core, 256 KBytes 2nd level cache per core, and a 32 MByte 3rd level cache shared per chip. 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 [6] for communication. The Hub chip is connected with the four POWER7 chips using four links, of 24GB/s each.

All runs use one process per UPC thread and schedule one UPC thread per POWER7 core. There are 32 UPC threads on each node and each UPC thread is bound to its own core. The results presented in this evaluation are the average of the execution time of five runs. The maximum execution time variation is less than 2% and occurs only in runs with a high number of UPC threads. Due to the characteristics of the Power 775 interconnect, the runs are isolated and no other task or job was running during the measurement. All benchmarks are compiled using the '`-qarch=pwr7 -qtune=pwr7 -O3 -qprefetch`' compiler flags. The evaluation tries to keep the computation constant per UPC threads (weak scaling). The evaluation doubles the dataset every quadruplication of UPC threads in the case of *Fish Grav* and *WaTor* benchmarks.

Table 1 presents the list of the benchmarks used in this evaluation and their communication pattern — they all use blocked data allocation. For this evaluation, five different binaries were generated for each program:

- *Baseline*: compiled with a dynamic number of threads and with the code transformations described in this paper disabled. The baseline

is the best available compiler and contains a number of optimizations: static coalescing [8], privatization [10], and remote updates [9].

- *Prefetch*: compiled with the inspector-executor code-transformation that prefetches and coalesces shared references at runtime [5].
- *Prefetch Optimized*: combines the inspector-executor transformation with the improvements presented, with dynamic number of threads.
- Hand-optimized: uses coarse-grained communication, manual pointer privatization, and collective communication whenever possible. This version also uses dynamic number of threads.
- *MPI*: contains coarse-grained communication and uses collective communication whenever possible. This version uses blocking communication and it does not use the one-side communication model introduced in MPI 2.0. One process is assigned to each core for the MPI implementation.

5. Experimental results

This experimental evaluation assesses the effectiveness of the transformations by presenting the following: (1) the performance on microbenchmarks to help understand the maximum speedup that can be achieved (2) the comparison with the sequential C version using strong scaling; (3) the performance of real applications; (4) the impact of the number of iterations examined; (5) an analysis of the overhead observed.

5.1. Microbenchmark Performance

Microbenchmarks are used to demonstrate the effectiveness of the code transformations. Indeed, the results presented in Figure 3 in log scale confirm that the code transformations are very effective when applied to the code that they target. In the `stream-like` benchmark the bandwidth increases close to linearly with the number of UPC threads for all versions including the baseline. The speedup due to the code transformations vary between 3.1x and 6.7x with the most significant gain due to prefetching. Adding static coalescing of struct fields to prefetching improves performance by 5-10%. The rightmost bars indicate that the overhead due to the insertion of inline checks is around 1%, which is bellow the measurement error. The inline-check code

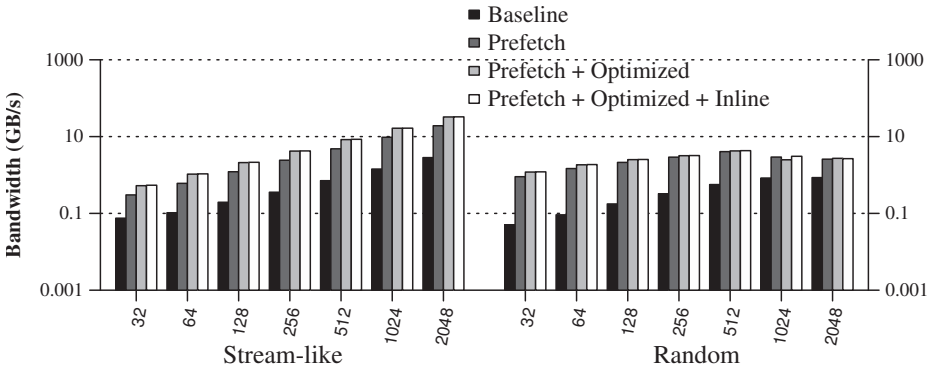


Figure 3: Performance in GB/s for the microbenchmark reading four fields from the same data structure for different versions.

Benchmark	Sequential C (gcc)	UPC Single-Thread		32 UPC Threads	256 UPC Threads
		Static	Dynamic		
Sobel	101.0	93.5	110.1	3.6	0.7
Fish	155.7	642.8	727.7	24.8	7.2
WaTor	9.8	48.3	791.3	98.2	28.1
MCop	16.6	19.2	29.3	140.8	22.0
Guppie	70.5	305.1	349.6	104.3	22.1

Table 2: Benchmarks compared with the serial C non-instrumented version and UPC version measured in seconds.

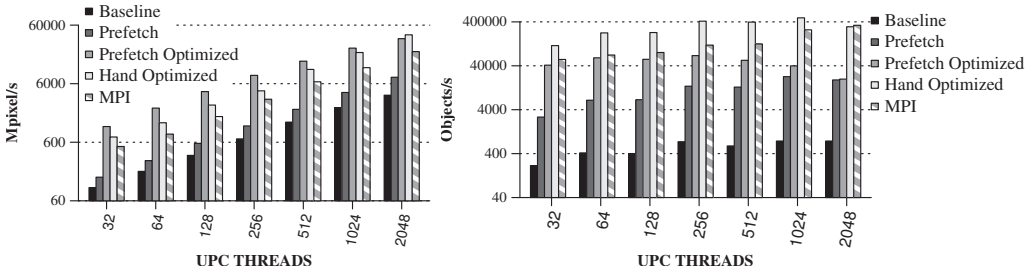
transformation inserts a branch before the entry points of the runtime, as described in Section 3.4.

The `stream-like` microbenchmark contains only remote shared references because it accesses shared data residing in the next-neighbour thread. Thus, the inline branch that avoids calls when the shared data are local is always taken. In contrast, the `random-access` microbenchmark results in a speedup between 3.2× and 21.6× from prefetch with an additional 4-8% due to struct-field coalescing. Furthermore, the inline transformation gives an additional 2-5% performance gain due to local shared access improvement.

An interesting observation emerging from the results is that `random-access` achieves better bandwidth than `stream-like` when the code transformations are enabled. This higher bandwidth results from the random traffic pattern in combination with the high-radix interconnect when using direct routes.

5.2. UPC Single-Threaded Slowdown

This section, prior to the scalability measurements, studies the performance of UPC language compared with the serial version. The single-thread



(a) (b)

Figure 4: Performance numbers for Sobel (a) and Fish Grav (b) benchmarks.

overhead, shown in Table 2 compares the execution time of the UPC version of the program running on a single thread with the execution time of a sequential C version of the code. The most important cause for the increase in the single-thread overhead is the use of *fat pointers* to reference data in distributed arrays. The run time in the single-threaded dynamic column in Table 2 is obtained running binaries where the number of threads is not known at compile time. When the number of the UPC threads is an integer power of two and it is known at compile time, then the pointer arithmetic call is replaced with shifts and masks. The runs with 32 and 256 UPC threads are performed with the inspector-executor transformation and the other code transformations presented in this paper. Some benchmarks, such as `Fish` and `Guppie`, run much slower than the C version even with a large number of threads because the compile-time data-access analysis is unable to detect and simplify accesses that are local.

The large slowdown for the dynamic single-threaded UPC version of `WaTor` can be explained by its large number of shared accesses for which the compiler generates calls to the runtime system. On the other hand, the smaller single-thread slowdown for `Guppie` can be explained by its irregular accesses that make its serial C version slower because of poor cache utilization. `Sobel` has the best potential compared with the other benchmarks for two reasons. First, it has good shared data locality because it fetches data only from the neighboring threads. Second, the code transformations removes the calls from the inspector and executor loops. The low performance in the single-thread version occurs because the program executes the unoptimized version of the loop to avoid the overhead of the shared-access analysis. Finally, the `MCop` benchmark is slower with 32 UPC threads than with one thread. This is expected because of the instrumentation code. On the other hand, the version with 256 UPC threads is faster than the one thread because

(i) the compiler creates fewer runtime calls and (ii) this section uses strong scaling. This slowdown underlines the key role of the code transformations that remove unneeded runtime calls automatically. Those results indicate that PGAS programmers and compilers should not focus only on reducing the cost of communication, but also in reducing the runtime calls.

5.3. Application Performance

This section explores the performance of the code transformations when applied to benchmarks. As described in Table 1, the access-analysis leads only to the removal of the runtime calls in **Sobel** and **Fish** benchmarks because those are the only benchmarks that contain regular accesses. The analysis allows the partial removal of the calls in **MCop**. **WaTor** and **Guppie** have complex access patterns and the compiler uses struct-field coalescing and the vector collection of elements in the inspector loop. The inline code transformation is also applied to the **MCop**, **WaTor**, and **Guppie** benchmarks.

Sobel achieves a performance gain between 1.5× and 2× using the inspector-executor (*prefetch*) code transformation as shown in Figure 4(a). The *prefetch optimized* technique achieves from 9.2× up to 12.3× speedup over the *baseline* because it allows for the complete removal of library calls. The *hand optimized* UPC version is faster than the *MPI* version because it uses one-side communication. However, the performance of the *hand optimized* and the *MPI* versions are converging with more than 256 UPC threads. One interesting observation is that the *prefetched optimized* version is faster than the UPC hand-optimized because of double buffering. The current version of the UPC language does not support asynchronous `memget/memput` calls. Thus, the exploitation of the overlapping communication and computation is the main advantage of the compiler transformation.

The **Fish** benchmark exhibits high performance gains because the *baseline* is inefficient, as shown in Figure 4(b). The compiler uses the CSLMADs representation to remove the runtime calls from the inspector and executor loops. The benchmark achieves from 40% up to 80% of the performance of the *hand optimized* version of the benchmark. The compiler successfully transforms one out of the two loops that contain fine-grained communication. The second loop implements a data reduction and becomes the bottleneck after the compiler applies the loop transformations.

The performance gain of the **WaTor** benchmark is lower than the **Sobel** and **Fish**: the *prefetch optimized* version is 1.12× to 1.72× faster than the

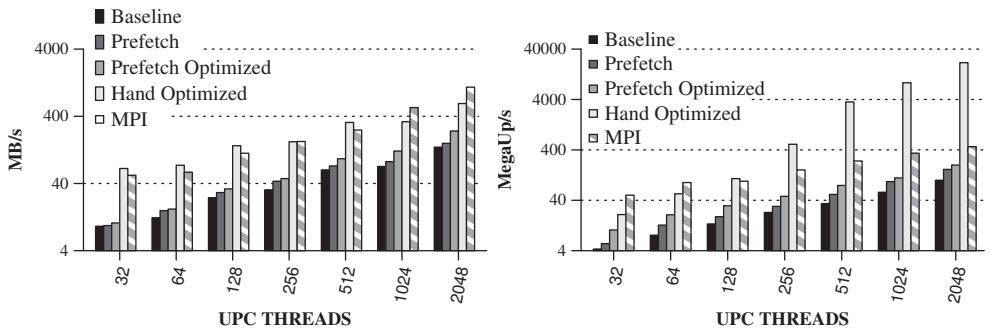


Figure 5: Performance numbers for the WaTor (a) and Guppie (b) benchmarks.

baseline (Figure 5(a)). The compiler transforms a loop structure that has constant number of iterations (25): the stencil computation. The compiler improves the performance of the remaining fine-grained shared accesses using the remote update to eliminate runtime calls [9]. The *MPI* version is faster but requires additional code before and after the calculation of force to move objects.

The **Guppie** benchmark uses random remote updates across a large shared array and calculates the performance in MegaUpdates/s. Due to irregular accesses, the *prefetch optimized* version of the benchmark achieves between 1.6× and 2.53× speedup over the *baseline* (Figure 5(b)). The compiler removes the calls from the inspector loops, thus decreasing the instrumentation overhead of collecting shared accesses. It is known that manual code modifications to this benchmark allow the application of the remote-update code transformations [9]. The benchmark uses a temporary buffer to fetch the data, modify, and write them back. The typical size of this buffer is 512 elements. In the UPC *hand optimized* version the number of elements is set to one. Thus, the compiler collapses the loops to apply the remote-update optimization and to exploit the hardware acceleration.

The *MPI* version of the **Guppie benchmark** generates the data on all processors and distributes the global table uniformly to achieve load balancing. The benchmark sends the addresses to the appropriate processors and the local process performs the updates. The MPI version is faster than the UPC versions for small number of threads. On the other hand, the manual UPC optimized version is 46× times faster than the MPI version running with 2048 Threads. This result provides strong evidence in support of the importance of the remote update code transformation [9]. The automatic compiler-optimized version achieves from 22% up to 48% the speed of the MPI version.

The *prefetch* code transformation gives a speedup from 1.6× up to 2.6× compared with the baseline version in the **MCop** benchmark, as shown in Figure 5.3. Applying the code transformations and manually unrolling the loops by four gives a speedup from 4.9× up to 6.3×. Despite the removal of most calls in the loops, there are still irregular references. The *manually optimized* version still contains irregular remote shared references. Prefetching these references improves the performance of the application. The *hand optimized* combined with the prefetching is two orders of magnitude faster than the MPI version.

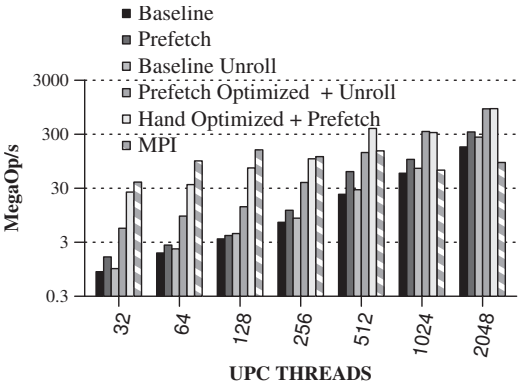


Figure 6: MCop benchmark performance.

5.4. Parameter exploration

An important question is how many iterations the runtime should prefetch. Previous studies of the inspector-executor code transformations [4, 20] suggest that this number should not be large. The main problem with inspecting and prefetching too many iterations is that the prefetching interferes with the normal operation of the various levels of caching. Figures 7(a) and 7(b) present one curve for the speedup and three curves for the cache miss ratios for different cache levels using different number of prefetched iterations (*Prefetch Factor*). The runs use 128 UPC threads (four nodes) and the IBM HPC Toolkit to obtain the performance-counter values. **Sobel** uses a 262144×262144 image (530 MB/ UPC Thread) and 0.32% of the total accesses are remote. Guppie uses 134 MB per UPC thread, and each UPC thread makes 16777216 updates. In total 99.21% of the shared accesses are remote. Each point in the horizontal axis is identified by the number of processing nodes and the average number of aggregated messages (inside parenthesis). The number of aggregated messages in **Sobel** is the number of iterations to inspect plus two for the borders of the image. On the other hand the number of messages aggregated in **Guppie** is calculated using the equation: $\#Iterations\_to\_inspect/UPC\_THREADS$, because of the random distribution.

For **Sobel**, the results indicate a correlation between the speedup and the cache misses at different levels of the memory hierarchy. The speedup cor-

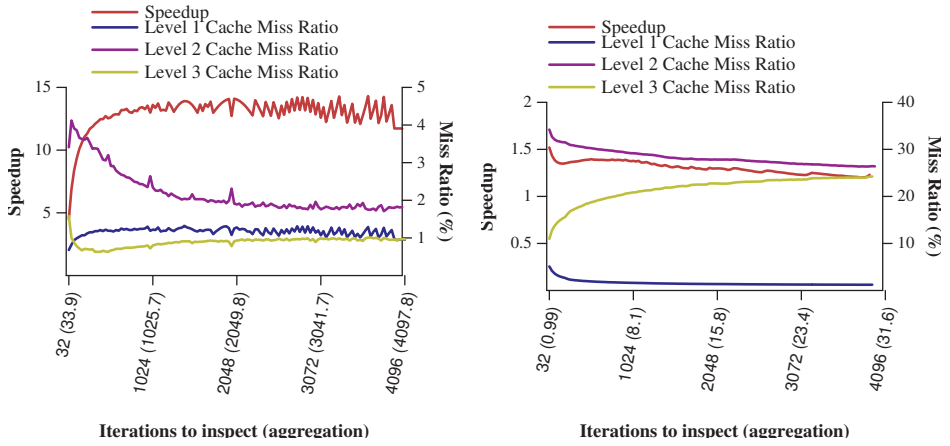


Figure 7: Speedup and cache misses for `Sobel` (a) and `Guppie` (b) using different number of iterations to inspect and average aggregation levels.

relates with the level two (L2) and level three (L3) cache misses. The most important observation is that the speedup remains constant when inspecting more than 672 iterations. Thus, when the compiler removes completely the instrumentation code, the limit of the application is not the network communication as next section explains.

On the other hand, **Guppie** incurs high miss ratio due to the random access pattern that it employs. In this case, despite the larger number of aggregated messages when we increase the number of elements to inspect, the speedup decreases for two reasons: (1) because there is an increase in the L3 miss ratio; and (2) because the cost of translating the shared addresses to local pointers increases with the number of prefetched elements. The runtime translates the shared index of the shared pointer into the index of the local buffer using a binary search algorithm because the fetched elements are not in the correct order. Thus, when the compiler does not remove completely the runtime calls using the CSLMADs, the performance gain should decrease as the number of iterations to be inspected increases.

5.5. Overhead Analysis

Two representative benchmarks are selected for the overhead evaluation: **Sobel**, which contains regular access and **Guppie**, which contains random accesses. Figure 8 presents a breakdown of the normalized execution time before and after the code transformations, using Linux Perf Tool. Sobel has regular access pattern and the compiler removes completely the calls from

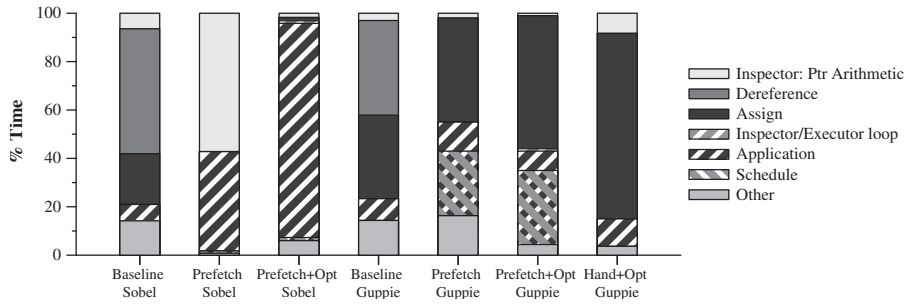


Figure 8: Normalized execution time breakdown using 32 UPC threads.

the inspector and executor loops. On the other hand, Guppie benchmark contains totally random access over the shared array. Using the inspector-executor approach (*Prefetch*) in **Sobel** benchmark, the time devoted to the computation decreases significantly. The **Sobel** benchmark spends more than 55% of the time in the shared-pointer arithmetic in the Prefetch version because of the additional calls in the inspector loops. The shared-pointer arithmetic translates the shared offset to the relative offset inside the thread. Removing the calls from the inspector and executor loops decreases the overhead to less than 8% of the application time.

On the other hand, the impact of the code transformations in **Guppie** is less than in **Sobel**. The optimized inspector-executor transformation in **Guppie** removes the calls from the inspector loops, but retains the calls in the executor loops. The improved inspector-executor transformation (*Prefetch Optimized*) reduces the communication overhead down to 57%. However, the overhead is transferred to the shared-reference analysis because of the irregular communication pattern. Therefore, the improved transformation successfully eliminates the apparent overhead in the application's code, but the runtime still processes the elements one by one. The inline code transformation has minor impact on the achieved performance and it is only visible for lower number of threads, when a certain portion of shared accesses are local. The *hand optimized* version of **Guppie** benchmark spends more time on assign that is replaced by the remote update runtime call.

5.6. Summary and Discussion

The code transformations presented can generate code for applications with regular accesses that achieve between 60% and 180% of the performance of the hand-optimized UPC version. The evaluation results support the argument that code transformations should focus on removing run-time calls completely in addition to the traditional compile and runtime optimizations.

On the other hand, there is still room for improvement for benchmarks that contain irregular communication.

6. Related Work

Shared Object Coalescing: Coalescing accesses that target a single shared object on the same UPC thread, or same remote node, is a well-known code transformation that aims to reduce instrumentation code. When using static analysis for data coalescing in Unified Parallel C [14, 8] and High Performance Fortran [13, 25], the compiler identifies, through data and control flow analysis, shared accesses to specific threads and creates a single run-time call to access multiple data items from the same thread. However, existing solutions do not completely remove the calls.

Shared-Pointer Privatization: The compiler uses information provided by the affinity expression of an `upc_forall` loop to privatize shared accesses to the local partition of the memory [10, 15]. For instance, an affinity expression that is pointer-to-shared usually indicates that the references are to local memory. Thus, the compiler can transform a *fat* shared pointer into a *thin* private pointer and completely remove the run-time call. Unfortunately, this approach only works for `upc_forall` construct and requires that the physical data placement be known at compile time.

Array Accesses Analysis: Linear Memory Access Descriptors (LMADs) [32] are a well-known representation that describes linear accesses to an array. LMADs are used for array-access analysis, coalescing of accesses, and for privatizing array accesses on various platforms. For instance, Xhu *et al.* use the LMAD representation to translate programs manually for distributed-shared-memory (DSM) systems [40]. Xunhao Li [28] and Garg [24] use a subset of LMAD called Restricted Constant Strided Linear Memory Access Descriptor (RCSLMAD) to identify memory locations of accessed array elements in Graphic Processor Units (GPUs).

Inspector-executor transformation: The inspector-executor strategy was initially developed to make communication more efficient in irregular applications [20] but it is now also a well-know code-transformation technique for PGAS languages. There are approaches [27] for compiler support using a global-name space programming model, or language-targeted optimizations such as: High Performance Fortran [11, 38], Titanium language [35], Unified Parallel C [4, 5], X10 [22], and Chapel [34]. Unfortunately, it turns out that even for loops with irregular shared references, the overhead is too high.

In contrast, the new approach presented in this paper applies aggressive transformations to remove calls from the inspector and executor loops and thus reduce its overhead.

7. Conclusion and Future Work

Eliminating runtime calls inserted by the compiler to translate shared-memory accesses into inter-node data transfers is essential to deliver reasonable performance in PGAS programming models executing in distributed-shared-memory machines. This paper uses a combination of new code transformations and adaptations of known techniques to the PGAS paradigm to eliminate such calls. The experimental evaluation indicates that when such calls are completely removed, the performance of automatically generated code is similar to the performance of coarse-grained versions of the benchmarks. On the other hand, there is room to improve the performance of applications that contain irregular accesses. The transformation helps to deliver the promised programming productivity of PGAS languages by allowing the programmer to write simpler fine-grained code.

There are still some aspects that we would like to investigate in the future. First of all, the existence of runtime calls inside the loops decreases the opportunities for optimizing the loop. Inter-procedural analysis can give additional information about the loop analysis optimizations to increase the opportunities of optimizing loops. Secondly, there are two applications categories that this paper doesn't examine. The first category is the applications that contain graph traversing. In this case the loops are not normalized and usually the traverse of the loop leads to irregular access pattern. Another category is the benchmark that contain sparse matrices, where the inspector-executor optimization fetches only the first level of the accesses.

Acknowledgments

The researchers at Universitat Politècnica de Catalunya and Barcelona Supercomputing Center are supported by the IBM Centers for Advanced Studies Fellowship (CAS2012-069), and the Spanish Ministry of Science and Innovation (TIN2007-60625, TIN2015-65316-P, and CSD2007-00050). 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 supported by the NSERC Collaborative Research and Development program of Canada.

References

[1] Aarseth, S., 2003. Gravitational N-Body Simulations: Tools and Algorithms. Cambridge Monographs on Mathematical Physics. Cambridge University Press.

[2] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Jr., G. L. S., Tobin-Hochstadt, S., March 2008. The Fortress Language Specification Version 1.0.

[3] Alvanos, M., Amaral, J. N., Tiotto, E., Farreras, M., Martorell, X., 2014. Reducing Compiler-Inserted Instrumentation in Unified-Parallel-C Code Generation. In: IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).

[4] Alvanos, M., Farreras, M., Tiotto, E., Amaral, J. N., Martorell, X., 2013. Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC. In: Proceedings of the 27th International Conference on Supercomputing. ICS '13.

[5] Alvanos, M., Farreras, M., Tiotto, E., Martorell, X., 2012. Automatic Communication Coalescing for Irregular Computations in UPC Language. In: Conference of the Center for Advanced Studies (CASCON).

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

[7] Barnaby Dalton and Gabriel Tanase and Michail Alvanos and George Almasi and Ettore Tiotto, 2014. Memory Management Techniques for Exploiting RDMA in PGAS Languages. In: In Workshop on Languages and Compilers and Parallel Computing (LCPC).

[8] Barton, C., Almasi, G., Farreras, M., Amaral, J. N., 2009. A Unified Parallel C compiler that implements automatic communication coalescing. In: 14th Workshop on Compilers for Parallel Computing.

[9] Barton, C., Cascaval, C., Almasi, G., Zheng, Y., Farreras, M., Chatterje, S., Amaral, J. N., June 2006. Shared memory programming for

large scale machines. Programming Language Design and Implementation (PLDI), 108–117.

[10] Barton, C. M., 2009. Improving access to shared data in a Partitioned Global Address Space programming model. Ph.D. thesis, University of Alberta.

[11] Brezany, P., Gerndt, M., Sipkova, V., 1994. SVM Support in the Vienna Fortran Compilation System. Tech. rep., KFA Juelich, KFA-ZAM-IB-9401.

[12] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V., Oct. 2005. X10: an object-oriented approach to non-uniform cluster computing 40 (10), 519–538.

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

[14] Chen, C. I. W., Yelick, K., 2005. Communication optimizations for fine-grained upc applications. In: In 14th International Conference on Parallel Architectures and Compilation Techniques. pp. 267–278.

[15] Chen, W.-Y., Dec 2007. Optimizing partitioned global address space programs for cluster architectures. Ph.D. thesis, University of California, Berkeley.

[16] Consortium, U., 2005. UPC Specifications, v1.2. Tech. rep., Lawrence Berkeley National Lab Tech Report LBNL-59208.

[17] Cormen, T. H., Stein, C., Rivest, R. L., Leiserson, C. E., 2001. Introduction to Algorithms, 2nd Edition. McGraw-Hill Higher Education.

[18] Cray Inc, April 2011. Chapel Language Specification Version 0.8.

[19] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., October 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13, 451–490.

[20] Das, R., Uysal, M., Saltz, J., shin Hwang, Y., 1993. Communication optimizations for irregular scientific computations on distributed memory architectures. Journal of Parallel and Distributed Computing 22, 462–479.

[21] Dewdney, A. K., 1984. Computer recreations sharks and fish wage an ecological war on the toroidal planet wa-tor. Scientific American, 14–22.

[22] Ebcioglu, K., Saraswat, V., Sarkar, V., 2004. X10: Programming for hierarchical parallelism and non-uniform data access. In: Proceedings of the International Workshop on Language Runtimes, OOPSLA.

[23] El-Ghazawi, T., Cantonnet, F., 2002. UPC performance and potential: a NPB experimental study. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing. pp. 1–26.

[24] Garg, R., Amaral, J. N., 2010. Compiling python to a hybrid execution environment. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 19–30.

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

[26] Kalla, R., Sinharoy, B., Starke, W., Floyd, M., 2010. Power7: IBM's Next-Generation Server Processor. Micro, IEEE 30 (2), 7 –15.

[27] Koelbel, C., Mehrotra, P., 1991. Compiling Global Name-Space Parallel Loops for Distributed Execution. IEEE Transaction Parallel Distributed Systems 2.

[28] Li, X., 2010. Jit4OpenCL: A Compiler from Python to OpenCL. Master's thesis, University of Alberta.

[29] Luszczek, P. R., Bailey, D. H., Dongarra, J. J., Kepner, J., Lucas, R. F., Rabenseifner, R., Takahashi, D., 2006. The HPC Challenge (HPCC) benchmark suite. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. SC '06. ACM.

[30] MPI Forum, 2014. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org`.

[31] Numwich, R., Reid, J., 1998. Co-array fortran for parallel programming. Tech. rep.

[32] Paek, Y., Hoeflinger, J., Padua, D. A., 2002. Efficient and Precise Array Access Analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) 24 (1), 65–109.

[33] Rajamony, R., Arimilli, L., Gildea, K., 2011. PERCS: The IBM POWER7-IH high-performance computing system. IBM Journal of Research and Development 55 (3), 3–1.

[34] Sanz, A., Asenjo, R., Lopez, J., Larrosa, R., Navarro, A., Litvinov, V., Choi, S.-E., Chamberlain, B. L., 2012. Global data re-allocation via communication aggregation in Chapel. In: SBAC-PAD.

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

[36] Tanase, G., Almási, G., Tiotto, E., Alvanos, M., Ly, A., Daltonn, B., 2013. Performance Analysis of the IBM XL UPC on the PERCS Architecture. Tech. rep., RC25360.

[37] Yelick, K. A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P. N., Graham, S. L., Gay, D., Colella, P., Aiken, A., 1998. Titanium: A High-performance Java Dialect. Concurrency - Practice and Experience 10 (11-13), 825–836.

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

[39] Zhang, Z., Savant, J., Seidel, S., 2006. A UPC Runtime System Based on MPI and POSIX Threads. Parallel, Distributed, and Network-Based Processing, Euromicro Conference on 0, 195–202.

[40] Zhu, J., Hoeflinger, J., Padua, D., 2003. Compiling for a hybrid programming model using the lmad representation. In: Languages and Compilers for Parallel Computing. Springer, pp. 321–335.