Optimization techniques for fine-grained communication in PGAS environments

Michail Alvanos

Submitted in in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Architecture

Universitat Politècnica de Catalunya
October 2013
Barcelona, Spain

Department of Computer Architecture
Universitat Politècnica de Catalunya

Optimization techniques for fine-grained communication in PGAS environments

Author: Michail Alvanos

Supervisors : Xavier Martorell
Associate Professor
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

José Nelson Amaral
Professor
Department of Computing Science
University of Alberta

Montse Farreras
Professor Collaborator
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Barcelona, August 2013

Abstract

Partitioned Global Address Space (PGAS) languages promise to deliver improved programmer productivity and good performance in large-scale parallel machines. However, adequate performance for applications that rely on fine-grained communication without compromising their programmability is difficult to achieve. Manual or compiler assistance code optimization is required in order to avoid fine-grained accesses. The downsides to manually applying code transformations are the increased program complexity and hindrance of the programmer's productivity. On the other hand, compiler optimizations of fine-grained accesses require knowledge of physical data mapping and the use of parallel loop constructs.

This thesis presents optimizations for solving the three main challenges of the fine-grained communication: (i) low network communication efficiency; (ii) large number of runtime calls; and (iii) network hotspot creation for the non-uniform distribution of network communication. To solve these problems, the dissertation presents three candidate solutions. It presents first an inspector-executor transformation for improving the network efficiency through runtime aggregation. Second, it presents incremental optimizations to the inspector-executor loop transformation to automatically remove the runtime calls. Finally, the thesis presents a loop scheduling transformation for avoiding network hotspots and the over-subscription of nodes. In contrast to previous work that use static coalescing, prefetching, limited privatization, and caching, the solutions presented in this thesis focus cover all aspects of fine-grained communication, including reducing the number of calls generated by the compiler and minimizing the overhead of the inspector-executor optimization.

A performance evaluation that uses various microbenchmarks and benchmarks, and presenting scaling and absolute performance numbers of a Power 775 machine, indicates that applications with regular accesses can achieve up to 180% of the performance of hand-optimized versions. In contrast, the transformations yield from 1.12X up to 6.3X speedup in applications with irregular accesses. The loop scheduling shows performance gains between +3% and +25% for NAS FT and bucket-sort benchmarks, and up to 3.4X speedup for the microbenchmarks.

Acknowledgments

When you set sail for Ithaca, wish for the road to be long, full of adventures, full of knowledge. – Ithaca, Constantine P. Cavafy

During the journey of the thesis, I have been fortunate to enjoy the advice, support, and friendship of a number of extraordinary people. I would like to thank each and every one of them. First of all, i would like to thank my supervisors Xavier Martorell, Montse Farreras, and José Nelson Amaral for their assistance and guidance during this work. They have offered me guidance and support in various ways. I started working with Xavier Martorell and Montse Farreras in the first stages of the thesis. Later Nelson offered some help with the compiler aspects of one paper and from that moment he stayed until the end. He acted as a catalyst during the process of the thesis.

I would like to thank many people from the IBM Toronto software lab and IBM T.J. Watson Research Center. First of all, i would like to thank Ettore Tiotto. Despite the time constraints and all the burden of managing the XL UPC compiler and runtime group, he was always available to help, not only for the administrative work, but also for technical level. His advices saved precious time and effort and provided guidance for the fast completion of the project. Furthermore, i would like to thank Nancy Wang for testing of the code and reporting bugs. Yaxun Liu for the setting environment and running the benchmarks in large scale. Gheorge Almasi, Ilie Gabriel Tanase, and Barnaby Dalton for their patience and support during the runtime developing. We had plenty and helpful discussions with Ilie Gabriel Tanase regarding the performance of the XL UPC compiler, especially in the last part of the thesis. I would like also to thank Patricia Clark and Debra A Domack for the access to Power 775 machines. Anny Ly for the interesting discussions about the performance of the Power 775 machines. Yaoqing Gao for reviewing the papers due to Canada Lab Publishing Process. I would like to thanks the people working on the Center of Advanced Studies (CAS) that helped during my stage on IBM Toronto: Jimmy Lo, Debbie Kilbride, and Emillia Tung. I would like to extend my gratitude to all the people of the IBM Toronto Software laboratory and IBM T.J Watson Research Center that help me during the various stages of this work.

I would like also to thank the IBM CAS students and Canadian friends: Carolina Simoes Gomes, Maria Attarian, Marios Fokaefs, Bo Wu, Joan Guisado Gòmez, Thomas Reidemeister, Norha M. Villegas, Michalis Athanasopoulos, and Mary Christmas (Maria Tsimpoukelli). These people make my stay in IBM Canada easier by having a lot of fun. My friends in Barcelona Supercomputing Center that all surrounded these years: Ivan Tanasic, Javier Cabezas, Marc Jorda, Lluc Alvarez, Lluis Vilanova, Thomas Grass, Xavier Teruel, Javier Bueno,

and Leonidas Kosmidis. Special thanks to Thanos Makatos, Dagalaki Efsevia, Vicky Antoniadou, George Nikiforos, Kallia Chronaki, Nohelia Meza, and Angely Bahamón because they were there when needed them most! I would also like to thank all of my old friends for the encouragement: Matthaios Kavalakis, Yannis Klonatos, Nikiforos Manalis, Billy Vassilaras, John Aparadektos Manousakis, Christos Margiolas, Maria Zaharaki, and Maria Psaraki. After so many years and living in different parts of the world, we always find some time to speak to each other! Last but not least, I would like to thank my family for their support in many aspects.

This work is supported by the IBM Centers for Advanced Studies Fellowship contracts no.CAS2011-057 / CAS2012-069 / CAS2013-12, by the Spanish Ministry of Science and Innovation through the grant CAN13001-779, and the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding agencies.

Michail Alvanos
Barcelona, 2013

Στην μνήμη της γιαγιάς μου Νικολέτας.
Στην μνήμη του Ιωάννη Γιαννέζου.
Στους γονείς μου Νικολαος και Χρυσοπηγή.
Στον αδερφό μου Ιωάννη.

To the memory of my grandmother Nikoleta.
To the memory of Ioannis Yiannezos.
To my parents Nikolaos and Chrysopigi.
To my brother John.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

BF Blocking Factor

CPU Central Processing Unit

CSLMAD Constant-Stride Linear Memory Access Descriptor

DSM Distributed Memory System

FFT Fast Fourier Transformation

GAS Global Address Space

GPU Graphics Processing Unit

LMAD Linear Memory Access Descriptor

MPI Message Passing Interface

PF Prefetch Factor

PGAS Partitioned Global Address Space Languages

SVD Shared Variable Directory

TPO Toronto Portable Optimizer

UPC Unified Parallel C

Chapter 1

Introduction

Next-generation architectures and large-scale parallel machines are increasing in size and in complexity. In this context, programming productivity is becoming crucial for software developers. Parallel languages and programming models need to provide simple means for developing applications that can run on parallel systems without sacrificing performance. New programming models provide attractive alternative ways of programming complex and parallel architectures. The programming model should provide a transparent method for effectively moving data without any efficiency loss relative to memory management.

The "automatic" memory management is becoming increasingly important for the compiler developer. The memory management must be transparent to the programmer, in a similar manner as the virtual memory in modern computers. High performance machines with attractive performance/price ratio are not enough to attract talented programmers into the High Performance Computing (HPC) due to complex programming. The productivity gap between distributed memory and shared memory machines is increasing especially with the standardization of the OpenMP [1] programming model.

A popular programming model for distributed systems is the Distributed Shared Memory systems (DSMs) [2, 3, 4, 5, 6]. 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 DSM systems rely on the page fault mechanism with page prefetching and often have poor performance on fine-grained communication [7]. Partitioned Global Address Space (PGAS) [8, 9, 10, 11, 12, 13, 14, 15] languages introduce some complexity through the partitioning of data, due to low performance of the Global Address Space languages and software distributed shared memory systems. PGAS languages extend existing languages or create new ones with constructs to express parallelism and data distribution.

Partitioned Global Address Space programming languages provide a uniform programming model for local, shared and distributed memory hardware. The programmer sees a single coherent shared address space, where variables may be

directly read and written by any thread, but each variable is physically associated with a single thread. These languages provide a simple, shared-memory-like programming model, where the address space is partitioned and the programmer has control over the data layout. They boost programmer productivity by using shared variables for inter-process communication instead of message passing [16, 17, 18].

1.1 Objective of the thesis

The key insight that motivates PGAS languages is that accessing data using individual reads and writes to the shared space, just as any programmer would do in a serial application increases programmer productivity. In a distributed environment, however, this coding style translates into fine-grained communication, which has poor efficiency and hinders performance of PGAS applications. The low communication efficiency of fine-grained data accesses has been identified by many researchers [19, 20] as one of the main bottlenecks of PGAS languages.

Regardless of all research community efforts, the de facto programming model for distributed memory architectures is still the Message Passing Interface (MPI) [21]. 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 his MPI equivalent, often nullifying the ease-of-coding advantage of these languages. In PGAS languages, the programmer accesses the data using individual reads and writes to the shared space. However, in a distributed environment this coding style translates into fine-grained communication, which has poor efficiency and hinders performance of PGAS applications [22, 20]. Due to the poor performance of fine-grained accesses, PGAS programmers optimize their applications by using large data transfers, whenever possible.

This dissertation tries to answer the following question: *Can applications written in the UPC programming model provide comparable performance with the hand-tuned MPI versions ?* The objective of this thesis is to improve performance of applications that use fine-grained communication, without hindering its programmability. The optimizations achieve comparable performance to the equivalent manual optimized versions. The thesis presents compiler and runtime optimizations for solving the three major problems of the fine-grained communications: (i) low efficiency of network communication; (ii) large number of runtime calls; and (iii) network hotspot creation for the non-uniform distribution of network communication, especially in all-to-all patterns. The work was carried out Unified Parallel C (UPC), although it can be applied to other PGAS languages as well.

1.1.1 Dynamic Data Coalescing

The first part of the thesis provides solutions to the problem of low network efficiency created by fine-grained communication. The research community has

proposed various techniques to decrease the performance penalty of fine-grained communication. Previous approaches to optimize the inter-node communication include the use of inspector-executor transformation [23, 24, 25, 26, 27, 28], static coalescing [22, 29, 30, 31], limited privatization [32, 33], and software cache [34, 35]. However, the existing solutions have two important limitations: (i) they require knowledge of physical data mapping at compilation time or the usage of a worksharing construct; (ii) they incur high overheads at runtime.

The programmer must specify the number of threads, the number of processing nodes, and the data distribution at compile time in order the compiler to have knowledge of physical data mapping at compile time (i). Also, the programmer must use worksharing constructs (upc_forall in UPC) to help the compiler to optimize the accesses. Nevertheless, applications do not make extensive use of the worksharing construct in practice. Thus, a substantial number of shared accesses inside serial loops (for, $while$ ...) are not optimized. In the second case (ii), the inspector-loop optimizations and cache systems usually produce high overheads that burden the application performance.

The first part of the thesis presents a compiler optimization with the proper runtime support to tolerate the latency of fine-grained accesses, through prefetching and coalescing techniques [36]. The optimization uses loop transformations and runtime support, to increase the communication efficiency and tolerate network latencies. The transformation uses the inspector-executor approach [24, 23, 37, 26] 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 accesses.

The aggregated network communication has three advantages. (i) It amortizes the per-message overhead at the end-points over the large amount of data being sent. (ii) It amortizes the per-packet routing and header information. (iii) Large messages require only a single acknowledgement rather than one per message, which contributes to reduction of traffic, end-point overhead, and possible network contention.

However, this approach has two drawbacks. First, the inspector-executor transformation inserts additional overhead for analyzing and aggregating at runtime the shared data. To increase the efficiency of this optimization, the system uses double buffering techniques to amortize the overhead. The second source of overhead is the large number of runtime calls. The inspector-executor transformation injects more runtime calls for collecting and inspecting elements. Thus, the number of runtime calls is doubled. To solve this problem, the thesis proposes a number of compiler optimizations for removing the calls as we discuss in the following section.

1.1.2 Reducing the overhead of inspector-loops

The inspector-executor transformation increases the network efficiency using large messages over the network interconnect. However, the transformation has a big disadvantage. The optimization increases the instrumentation overhead due to

additional runtime calls in the inspector loop. The second part of the thesis, proposes different approaches to reduce the number of runtime calls. The thesis presents a set of compiler optimizations [38] to increase the efficiency of UPC language by removing runtime calls and addressing some of the weaknesses found in traditional compiler optimizations in loops. The techniques include inspector-executor improvements to decrease the overhead and a lightweight code invariant loop motion.

The first technique employed involves the usage of Constant Stride Linear Memory Address Descriptors (CSLMADs) [39], a restrictive form of Linear Memory Descriptors [40, 41]. CSLMADs constitute an efficient way to capture accurate array access information. Thus, the compiler can effectively remove the calls from the inspector and executor loops when the access pattern is regular. Furthermore, the compiler uses a temporary vector to collect the shared indexes in the inspector loops to improve performance on benchmarks with irregular access patterns.

The last part of this section presents a lightweight loop code motion. Loop-invariant code motion is a traditional compiler optimization which performs this movement automatically. The compiler automatically moves outside of the loop main body statements and expressions that don't affect the semantics of the program. However, shared scalar and pointers can make the analysis hard, leaving some shared accesses inside the body of the loop. The algorithm uses the reaching definitions analysis through the Static Single Assignment (SSA) [42] representation to detect and move the statements that use shared variables before the loop.

1.1.3 Improving the all-to-all communication pattern

The last part of the thesis presents optimizations for improving the network communication and avoiding the creation of hotspots. In order to effectively avoid congestion the programmer or the runtime must spread non-uniform traffic evenly over the different links. Moreover, an important number of UPC applications that require all-to-all communication can create hotspots during the communication without the awareness of the programmer. This pattern shows up in a large number of scientific applications including FFT, Sort, and Graph 500. The UPC programs can suffer from shared access conflicts on the same node and network link congestion.

This section first explores the possible approaches to distributing the accesses through the network by manually modifying the source code. Next, the thesis presents a loop transformation technique that improves the performance of the all-to-all communication without the programmer's interference. The transformation "skews" or randomly distributes the traffic using all the UPC threads to reduce the possibility of hotspot creation on the interconnection network and overwhelming of the nodes.

1.2 Contributions

This dissertation presents how new programming models provide comparable performance with the message passing models, when using fine-grained communication. The thesis makes the following contributions:

- It presents coalescing and prefetching techniques that use loop transformations and runtime support to increase communication efficiency and tolerate the network latencies [43, 36, 38]. It demonstrates that dynamic analysis and coalescing at runtime can significantly improve the performance of fine-grained accesses inside loops, without having to know the physical data mapping or use the *upc_forall* parallel structure.

- It proposes a number of optimizations to decrease the impact of library calls that fine-grained communication produces [38]. The optimizations include an analysis based on Constant-Stride Linear Memory Descriptors (CSLMADs), insertion of inline checks for local data, and a new shared-reference-aware loop-invariant code motion. This latter, is designed specifically for PGAS languages, to deliver improved performance for benchmarks that rely on fine-grained communication. This new approach to the compilation of PGAS applications results in programs containing fine-grained access benchmarks having comparable performance to that of coarse-grained benchmarks.

- It presents a compiler loop transformation that automatically schedules the loop iterations to increase the applications performance by decreasing the potential contention of the network [44, 45]. The evaluation shows that a compilers can provide comparable performance to the manual modified loop.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces the Unified Parallel C programming language and the IBM XL UPC compiler framework. Chapter 3 presents the hardware environment and the benchmarks used for the evaluation. Chapter 4 demonstrates the inspector-executor transformation and runtime dynamic aggregation of shared-objects. Chapter 5 presents compiler transformations to reduce the overhead of the automatically compiler created calls. Chapter 6 presents approaches and compiler transformations to improve the performance of network using loop scheduling. The related work is reviewed in Chapter 7. Chapter 8 draws conclusions based on the findings, and discusses future research plans.

Chapter 2

Background

2.1 Partitioned Global Address Space Languages

Partitioned Global Address Space Languages (PGAS) programming languages provide a uniform programming model for local, shared and distributed memory hardware. The programmer sees a single coherent shared address space, where variables may be directly read and written by any thread, but each variable is physically associated with a single thread. PGAS languages, such as Unified Parallel C [9], Co-Array Fortran [10], Fortress [11], Chapel [12], X10 [13], Global Arrays [46], and Titanium [14], extend existing languages with constructs to express parallelism and data distribution.

The message passing models, such as MPI [21], use library calls for explicit communication between processes. On the other hand, in PGAS languages, the programmer can access each variable between processes without knowing if the access is remote or local. Figure 2.1 presents the differences between the programming models. Example implementations of shared memory programming model are the OpenMP [1] and Pthreads [47]. The downside of this approach is that the programmer is not always aware of the locality of data and can use remote accesses that lead to performance degradation. Other popular alternative shared memory programming models are for the Software Distributed Memory Systems (DSMs)[2, 3], such as Nanos DSM [4], ThreadMarks [5] and ParaADE [6]. Similarly to the PGAS, the programmer sees a global address space and the runtime is responsible for the communication.

2.2 Unified Parallel C

The Unified Parallel C (UPC) [9] is an example of PGAS programming model as an extension of the C programming language [48] 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 processor.

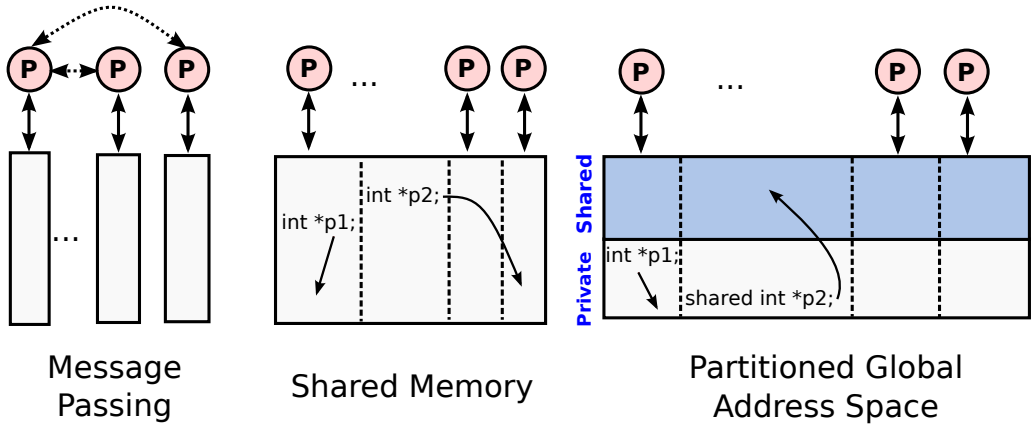

Figure 2.1: Comparison of parallel programming models

In all cases, the compiler translates the shared remote access to runtime calls for fetching and storing the data. The result is portable high-performance compilers that run on a large variety of shared and distributed memory multiprocessors.

In order to better understand the concept of UPC programming model, this section explains the basic C language extensions. In UPC, there are two new keywords, the keyword `'THREADS'` that describes the total number of threads and the keyword `'MYTHREAD'` that describes the thread identifier starting from 0. The language defines two different types of variables, the *shared* and the *private*. The shared variables must be explicitly declared by using the keyword shared or they are considered private. The shared variables are used for communication between threads because they are visible from any thread. The private variables can only be accessed by a single thread. To access shared variables, the programmer must specify the keyword `shared` to the pointer declaration.

```
1  int OUT[N][N];
2  int IN[N][N];
3
4  void stencil_kernel(){
5    int i,j;
6    for(i=1; i<N-1; 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 2.1: Serial version of a stencil kernel.

Furthermore, in UPC there is a new type of `for` loop, the `upc_forall` loop, in which the iterations are executed in parallel, according to the language semantics. Each thread executes a fraction of iterations concurrently with other threads. At

the header of the loop, there is an additional expression, called affinity expression. The affinity expression specifies which executions of the loop will be performed by a thread.

Listing 2.1 presents a serial version of a stencil benchmark. A straightforward UPC parallel version is shown in Listing 2.2. Arrays `IN` and `OUT` are declared as `shared` (lines 1-2), 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 2.2: Parallel version of a stencil kernel.

In this example access locality to array `OUT` is developed through the use of the affinity expression. Each runtime call may imply communication of one element of the array, leading to fine-grained communication which leads to poor performance. Communication traffic in this case is $O(N \times N)$ elements, with one access per element.

```
1  #define B N*(N/THREADS)
2  shared [B] int OUT[N][N];
3  shared [B] int IN[N][N];
4
5  void stencil_kernel(){
6    int i,j;
7    upc_forall(i=1; i<N-1; i++; &OUT[i] ){
8      for(j=1; j<N-1; j++ ){
9        OUT[i][j] = 0.25f * (IN[i-1][j] + IN[i+1][j] +
10                            IN[i][j-1] + IN[i][j+1] );
11     }
12   }
13 }
```

Listing 2.3: Blocked parallel version of a stencil kernel.

A better distribution of the shared data can be achieved through the use of layout modifiers. Listing 2.3 shows a distribution by rows, where each thread

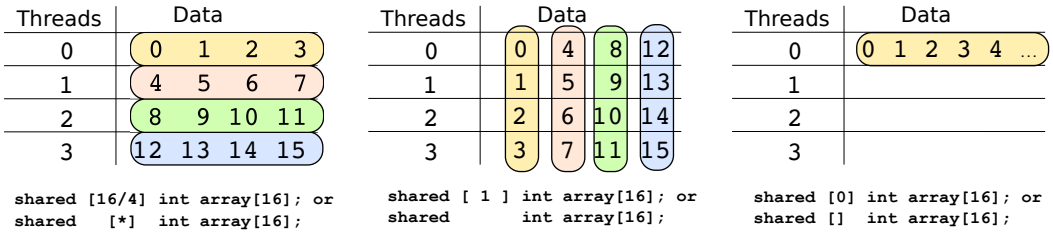

Figure 2.2: Comparison of different data blocking possibilities.

owns n consecutive rows. Where $n = N/THREADS$ as specified by the [B] blocking factor. The Blocking Factor (BF) is layout qualifier that dictates the number of successive elements placed on the same UPC thread. This feature gives the option to the programmer to select the data partitioning. The selection of proper blocking scheme can significantly affect the performance of the application. Programmers use two common blocking schemes: the 'blocked' or the 'cycled'. Figure 2.2 illustrates the different blocking schemes with the array declarations. Note that the programmer has also the possibility to allocate shared objects or arrays with affinity to only one UPC thread (Figure 2.2 right).

```
1  typedef struct {int r[N];} Row;
2  shared [*] Row IN[N+2*THREADS];
3  shared [*] Row OUT[N+2*THREADS];
4
5  void stencil_kernel(){
6    int i,j;
7    int lower, upper;
8    int rowspp;
9    rowspp = (N+2*THREADS)/THREADS;
10   lower = MYTHREAD*rowspp;
11   upper = (MYTHREAD+1)*rowspp -1;
12   /* exchange shadows - (code incomplete, no bounds check) */
13   /* shadow for lower boundary */
14   upc_memget(&IN[lower], &IN[lower-2], sizeof(Row));
15   /* shadow for upper boundary */
16   upc_memget(&IN[upper], &IN[upper+2], sizeof(Row));
17   for(i=lower; i<upper; i++){
18     for(j=1; j<N-1; j++ ){
19       OUT[i].r[j] = 0.25f * (IN[i-1].r[j] + IN[i+1].r[j] +
20                              IN[i].r[j-1] + IN[i].r[j+1]);
21     }
22   }
23 }
```

Listing 2.4: Stencil optimized parallel version.

In this example, the non-local read memory accesses are reduced to $O(2 \times$

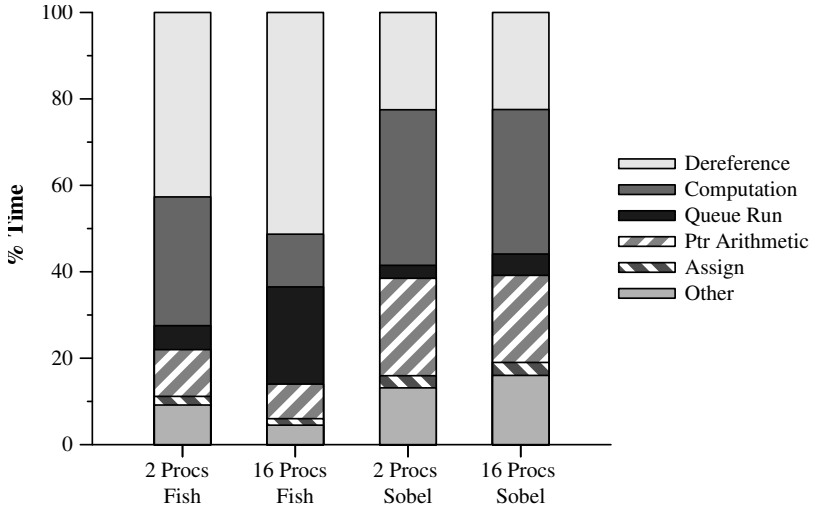

Figure 2.3: Normalized execution time breakdown of gravitational *fish* and *sobel* benchmarks using 2 and 16 processes.

$N \times THREADS$) because only computation of positions in the boundary of each thread data access remote data. However, one access per element is still performed, leading to fine-grained communication.

To avoid fine-grained communication the UPC language provides data movement primitives: upc_memget, upc_memput, and upc_memcpy. In listing 2.4, the lower and upper boundary rows of the IN array on each thread are copied into a shadow row which is local to that thread. The number of accesses decreases to $O(2 \times THREADS)$, each access bringing a row of size N. This code has better performance although, the programmability has been hindered. This contradicts the philosophy of PGAS languages: ease of programming and productivity.

2.3 Overheads of fine-grained accesses

When the physical data mapping is unknown at compiler time, the compiler does not apply most of the UPC specific optimizations. Two problems arise from these 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 the large number of runtime calls created.

Figure 2.3 presents the execution time breakdowns of two different benchmarks: *fish* [49] and *Sobel* [50]. It shows that little time is spent on the actual computation: 30% in *fish* and 35% in *Sobel* with 2 UPC threads. Several sources of overhead are identified: (i) time spent in accessing shared data (*dereference* phase) shows the impact of the communication latency for fine-grained accesses where communication is necessary for each dereference call; (ii) Shared pointer

arithmetic (*Ptr Arithmetic*) also has an impact since shared pointers contain more information than plain pointers and operations with them are expensive. This impact is greater in Sobel (20%) compared to gravitational fish because it contains nine shared accesses per loop iteration. Overall, the communication latency and the overhead from the large number of runtime calls burdens the performance of applications and motivates the presented optimization.

Listing 2.5, presents the computation kernel of the gravitational fish benchmark. The benchmark emulates fish movements based on gravity. The benchmark is an N-Body gravity simulation, using parallel ordinary differential equations [49]. Arrays `fish` and `accel` are declared as shared (lines 1-2). Shared arrays or shared objects are accessible from all UPC threads. The layout qualifier `[NFISH/THREADS]` specified that the shared object is distributed to different UPC threads in blocked form. 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 (`&fish[i]`) specifies that the owner thread of the specified element executes the ith loop iteration.

```
1  typedef struct fh { 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     /* Phase 1: Force calculation */
10    upc_forall (i=0; i<NFISH; ++i; &fish[i]) {
11       tmpx = tmpy = 0;
12       for (j = 0; j < NFISH; ++j) {
13          dx = fish[j].x - fish[i].x;
14          dy = fish[j].y - fish[i].y;
15          a = calculate_force(dx,dy);
16          tmpx += a * dx / r; tmpy += a * dy / r;
17       }
18       acc[i].ax = tmpx; acc[i].ay = tmpy;
19    }
20    upc_barrier ();
21    ...
22 }
```

Listing 2.5: UPC version of gravitational Fish.

Runtime calls are responsible for fetching, or modifying, the requested data. Each runtime call may imply communication of one element of the array, leading to fine-grained communication, which in turn leads to poor performance. The compiler transforms each shared access to runtime calls. Read accesses are translated into a `__ptr_deref` runtime call, while the write accesses translate into `__ptr_assign`. Each runtime call can have various arguments, including the offset

of shared variable and the element size. At its turn, internally a PGAS runtime takes care of accessing the data, which can be in a remote node and so it may imply communication, a *get* or a *put*, depending if you are reading or writing. Before the accessing shared pointers, the compiler also creates calls for shared pointer arithmetic (*__ptr_arithmetic*) as shown in Listing 2.6. The shared pointer is a fat pointer that contains information about the offset, the thread, and the allocated size. Note, that when the compiler knows the number of the UPC threads, it can eliminate the pointer arithmetic call and replace it with shifts and masks.

```
1  for each time step {
2    ...
3    /* Phase 1: Force calculation */
4    upc_forall (i=0; i<NFISH; ++i; &fish[i]) {
5       tmpx = tmpy = 0;
6       for (j = 0; j < NFISH; ++j) {
7          __ptr1 = __ptr_arithmetic( &fish[j].x, ...);
8          __tmp1 = __ptr_dereference( ptr1, ...);
9          __ptr2 = __ptr_arithmetic( &fish[j].y, ...);
10         __tmp2 = __ptr_dereference( ptr2, ...);
11         __ptr3 = __ptr_arithmetic( &fish[i].x, ...);
12         __tmp3 = __ptr_dereference( ptr3, ...);
13         __ptr4 = __ptr_arithmetic( &fish[i].y, ...);
14         __tmp4 = __ptr_dereference( ptr4, ...);
15
16         dx = __tmp1 - __tmp3; dy = __tmp2 - __tmp4;
17         a = calculate_force(dx,dy);
18         tmpx += a * dx / r; tmpy += a * dy / r;
19      }
20      __tmp5 = tmpx;
21      __ptr5 = __ptr_arithmetic( &acc[i].ax, ...);
22      __ptr_assign( &__tmp5, ptr5, ...);
23      __tmp6 = tmpy;
24      __ptr6 = __ptr_arithmetic( &acc[i].ay, ...);
25      __ptr_assign( &__tmp6, ptr6, ...);
26   }
27   upc_barrier ();
28   ...
29 }
30 ...
```

Listing 2.6: Final form of the source code.

The PGAS programming model boosts programmer productivity by providing shared variables for inter-process communication instead of message passing [16, 17, 18]. However, the performance of these emerging languages has room for improvement. Today, PGAS programs deliver scalable performance on clusters only when they are well written. In previous work, many researchers have pointed out the bottlenecks of PGAS languages [19, 20, 51]. Some reasons for the limited PGAS performance are:

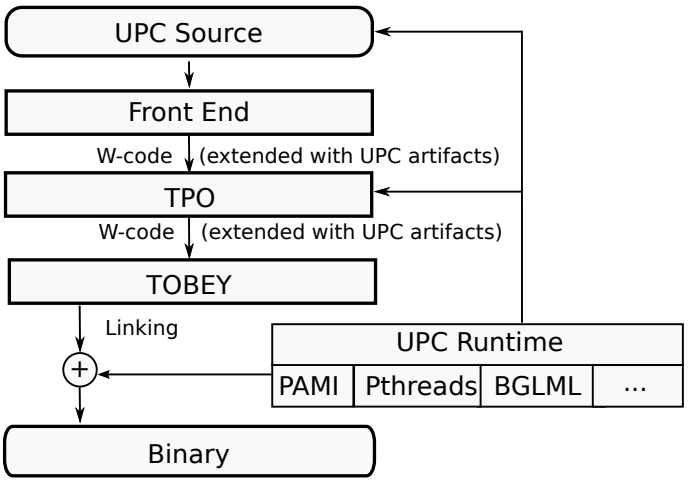

Figure 2.4: XL UPC compiler framework.

- The low communication efficiency of fine-grained accesses data accesses.
- The absence of non-blocking communication mechanisms on PGAS languages. Blocking communication mechanisms eliminate any possible opportunity for computation and communication overlap.
- The existence of function calls for accessing shared data and the absence of inter-procedural analysis. They disable common compiler loop optimizations because they reduce the scope of many data-flow optimizations such as copy propagation and common sub-expression elimination.
- Transparency of shared accesses leads to possible hotspot creation on the network.

2.4 The IBM UPC Compiler and Runtime System

This section presents the XL UPC compiler framework [52, 53] used to implement the optimization. The XL UPC compiler has three main components: (i) the `Front End (FE)`; (ii) the `Toronto Portable Optimizer (TPO)` high-level optimizer; (iii) a low-level optimizer (TOBEY). Figure 2.4 shows the role of each component in the compilation of UPC programs.

2.4.1 XL UPC Compiler Framework

The XL UPC compiler has three main components: (i) the `Front End (FE)` transforms the UPC source code to an intermediate representation (W-Code); (ii) the `Toronto Portable Optimizer (TPO)` high-level optimizer performs machine independent optimizations for UPC and C/C++ languages; (iii) and a low-level optimizer performs machine-dependent optimizations.

The compiler front end tokenizes and parses UPC source code, performs syntactic and semantic analysis, and diagnoses violations of the Unified Parallel C (v 1.2) language rules. It then generates an intermediate language representation (IR) of the UPC program, augmented with UPC extensions such as the layout of shared arrays, and the affinity expression of a `upc_forall` loop, for example. The augmented intermediate representation (W-Code + UPC extensions) is consumed by the high-level optimizer, or TPO (Toronto Portable Optimizer). The high-level optimizer component has been extended to perform UPC specific optimizations; it also performs a subset of the traditional control-flow, data-flow and loop optimizations designed for the C language on UPC source code.

The high-level optimizer interfaces with the PGAS runtime (or XL UPC runtime) through an internal API that is used to translate operations on shared objects such as dereferencing (reading/writing) a pointer to a shared object (such as a shared array for example) and performing pointer arithmetic operations. Finally, the high-level optimizer produces a modified version of the IR (Optimized Wcode) that lacks UPC-specific extensions (operations on UPC extensions are either translated to PGAS runtime calls or resolved to the base IR through optimizations). The IR produced by the high-level optimizer is consumed by the low-level optimizer (TOBEY), which performs further optimizations that are UPC unaware. After optimizing the low level IR, TOBEY generates machine code for the target architecture. This process is repeated for each compilation unit. To complete the compilation process, the XL UPC compiler invokes the system linker, which links compiler-generated objects and any required libraries (such as the PGAS runtime library) to form an executable.

The XL UPC compiler provides a number of UPC specific optimizations. The next part of this section examines the main high level UPC specific optimizations, which are implemented in the XL UPC compiler.

Shared object access optimizations

The XL UPC compiler implements a set of performance optimizations on shared array accesses. The compiler can partition shared array accesses performed in a `upc_forall` work-sharing loop into two categories: shared local accesses (accesses that have affinity with the issuing thread) and shared remote accesses. Shared array accesses that have been proven to have affinity with the issuing thread are optimized by the compiler in such a way as to eliminate unnecessary runtime calls. Shared array accesses that are remote can be coalesced by the compiler to reduce the communication latency.

Shared object access privatization

In a typical scenario, the XL UPC compiler translates accesses to shared arrays by generating an appropriate set of runtime function calls. In the context of a `upc_forall` loop the compiler can often prove that the memory read and/or

written during an array access operation resides in the local address space of the accessing thread; in such cases the compiler generates code that performs the indexing (pointer arithmetic) operations required to access the shared array directly. To do so, the compiler retrieves the address of the shared array partition in the local address space (the private address space of the accessing thread) via a runtime function call. It moves the runtime call outside the `upc_forall` loop nest containing the shared array access being translated. This is legal because the local shared array address is loop-invariant. Finally, the compiler uses the local array address to index the appropriate shared array element, doing any pointer arithmetic operations that are necessary locally [54]. Figure 2.5 presents an example of local shared accesses privatization.

Programmer's Code	After transformation

`#define SCAL 3.0`
 	
`shared double a [N] ;`
`shared double b [N] ;`
`shared double c [N] ;`
 	
`void StreamTriad(){`
`  int i ;`
`  upc_forall (i=0; i<N; i++; &a[i]){`
`  a[i] = b[i] + SCAL * c[i] ;`
`  }`
`}`
	`#define SCAL 3.0`
	`shared double a [N] ;`
	`shared double b [N] ;`
	`shared double c [N] ;`
	
	`void StreamTriad(){`
	`  int i ;`
	`  double *__aBase = __xlupc_base_addr(a) ;`
	`  double *__bBase = __xlupc_base_addr(b) ;`
	`  double *__cBase = __xlupc_base_addr(c) ;`
	
	`  upc_forall (i=0;i<N;i++; &a[i]){`
	`  *( __aBase + OFFSET(i)) =`
	`  *( __bBase + OFFSET(i)) +`
	`  SCAL * ( *( __cBase + OFFSET(i)))`
	`  }`
	`}`

Figure 2.5: Example of privatization optimization.

Shared object access coalescing

Normally, the XL UPC compiler translates a remote shared array access by generating a call to the appropriate runtime function. For example, reading from (or writing to) multiple shared array elements that have affinity to the same remote thread causes the compiler to generate a runtime call for each of the array elements read. In the context of a `upc_forall` loop nest, the compiler can often determine that several array elements are read from the same remote partition. In this case, the compiler combines the read operations and generates a single call to the runtime system to retrieve the necessary elements together, thus reducing the number of communication messages between the accessing thread and the thread that has affinity with the remote array partition [29].

Shared object remote updating

This optimization targets read-modify-write operations on a shared array element. When translating a read operation followed by a write operation on the same shared array element the compiler normally generates two runtime calls: one to retrieve the shared array element and one to write the modified value back. When the compiler can prove that the array elements being read and written have the same index, it can generate a single runtime call, with which it instructs the runtime system to perform the update operation on the thread that has affinity with the array elements accessed. This optimization reduces the number of calls required to translate the read-modify-write pattern from two to one, and therefore reduces reducing the communication requirement associated with the operation.

Array idiom recognition

Unified Parallel C programs often include loops that simply copy all elements of a shared array into a local array, or vice versa, or loops used to set all elements of a shared array with an initial value. The XL UPC compiler is able to detect these common initialization idioms and substitute the fine-grained communication in such loops with coarser-grained communication. The compiler achieves this goal by replacing the individual shared array accesses with calls to one of the UPC string handling functions: `upc_memget`, `upc_memset`, `upc_memcpy`, or `upc_memput`. Figure 2.6 provides an example of this optimization.

Programmer's Code

```
#define N 16384
#define BF (N/THREADS)

shared [BF] int a[N];
int b[N];

int main () {
 int i;
 if (MYTHREAD==0) {
    for (i=0; i<N; i++){
      a[i]=b[i];
    }
 }
}
```

After transformation

```
#define N 16384
#define BF (N/THREADS)

shared [BF] int a[N];
int b[N];

int main () {
  int i;
  if (MYTHREAD==0) {
    for (i=0; i<N; i+=BF) {
      upc_memput(&a[i], &b[i], BF*sizeof(b[i]));
    }
  }
}
```

Figure 2.6: Example using the array idiom recognition.

Parallel loop optimizations

The XL UPC compiler implements a set of optimizations that remove the overhead associated with the evaluation of the affinity expression in a `upc_forall` loop. The affinity expression in a `upc_forall` loop could be naively translated by using a branch to control the execution of the loop body. For example, an integer affinity expression "i" could be translated by inserting the conditional expression `(i == MYTHREAD)` around the `upc_forall` loop body. The compiler, on the other hand, translates the `upc_forall` loop into a for loop (or a for loop nest) using a strip-mining transformation technique that avoids the insertion of the affinity branch altogether, and removes a major obstacle to the parallel scalability of the loop. For example, the parallel loop:

```
upc_forall ( i =0; i<N; i ++; i ){
    a[i] = b[i] + scalar * c[i]
}
```

will be transformed to:

```
for ( i =0; i < N; i ++){
    if (( i % THREADS) == MYTHREAD )
        a[i] = b[i] + scalar * c[i] ;
}
```

However after the optimization the final loop will not contain the affinity branch:

```
for ( i =MYTHREAD; i<N; i +=THREADS){
    a[i] = b[i] + scalar * c[i] ;
}
```

2.4.2 Runtime

The PGAS runtime provides a platform-independent interface that allows the executable to run in different machines and it is designed for scalability in large parallel machines [55, 54]. Currently supported platforms include: shared-memory multiprocessors using the Pthreads library; Low-level Application Programming Interface [56]; and Parallel Active Messaging Interface (PAMI) [57]. Experimental ports of the UPC runtime also exists for other Messaging systems like Myrinet Express [58], BlueGene/L message layer [59], and Deep Computing Messaging Framework (DCMF) [60]. The PGAS runtime is designed for a hybrid mode of operation on clusters of SMP nodes. For example, UPC threads communicate through shared memory when possible, and they send messages to other nodes through one of several available transports. A similar approach was followed in the GASNet runtime system [61]. The runtime exposes to the compiler an Application Program Interface for managing shared data and synchronization.

Memory Management

In UPC, there are two types of memory allocations: *local memory* allocations performed using malloc and which are currently outside of the tracking capability of the runtime, and *shared memory* allocated using UPC specific constructs such as `upc_all_alloc()`, `upc_alloc()` and `upc_global_alloc()`.

Within the UPC specific family of allocation functions, the runtime currently employs a set of optimizations to help both with remote memory address inference with memory registration. The memory registration pins the memory (no memory swap and no page movement) and notifies the network interface controller (NIC) of the virtual-to-physical address mapping of this memory. The memory registration is necessary to enable the Remote Direct Memory Access (RDMA) mechanisms that are used in high performance machines. First, shared arrays allocated using `upc_all_alloc()` are allocated symmetrically at the same virtual memory address on all locations. In doing so, the XL UPC allocator creates a reserved area in the virtual address space of each process called the symmetric partition. The starting virtual address of each symmetric partition, called the origin, is identical across all threads and distributed shared arrays are then stored in blocks of memory located isomorphically in each symmetric partition. The READ and WRITE operations on elements of a shared array allocated with `upc_all_alloc`, therefore, know the virtual address in the remote locations. Shared memory allocated using the `upc_global_alloc` primitive is handled in a similar fashion.

The users can declare shared arrays allocated in one thread's address space using `upc_alloc`. In this situation, other threads can only obtain references to these arrays using another explicit data exchange. If such a reference is obtained, a remote thread can perform remote reads or writes to the data of the shared array. In the current XL UPC implementation, the memory allocated using `upc_alloc` is not registered with PAMI. If the `upc_memput` access or `upc_memget` access memory that explicitly allocated with system `malloc`, then the runtime uses active messages. For example, in `upc_memput` call, the destination must always be a shared memory array while the source can actually be from an array allocated with `malloc`.

Point to point data transfers

Point to point data transfers are employed by UPC whenever a remote array index is accessed or whenever `upc_memput`/`upc_memget` are invoked. The underlying network supports three modes of data transfer, which are referred to as active messages with short, large data, and RDMA. The XL UPC runtime system exploits all three modes.

The runtime enforces the ordering constraints imposed by the UPC memory consistency model. The underlying interconnect and PAMI library do not preserve the order of messages between a source and destination process. For this reason, when mapping UPC constructs to PAMI primitives, the runtime explicitly waits for a message to be remotely delivered before sending the next message to *the same*

destination. However, the runtime can send a message to a different destination without waiting for the acknowledgement of messages on previous destinations. While this can be seen as a performance bottleneck, in practice this decision did not affect the performance of most benchmarks. This is because often the threads send to different destinations before sending again to a particular one.

Atomic Operations

The XL Unified Parallel C compiler implements the atomic extension of the Unified Parallel C language as proposed by Berkeley UPC [62]. This extension, which is on track to being adopted as part of the UPC 1.3 specification, allows users to atomically read and write private and shared memory in a UPC program. With atomic functions, you can update variables within a synchronization phase without using a barrier. The atomic functions are included in List 2.7. The function prototypes have different variants depending on the values of `type, X`, and `RS`. `X` and `type` can take any pair of values in (I, int), (UI, unsigned int), (L, long), (UL, unsigned long), (I64,int64_t), (U64,uint64_t), (I32,int32_t), (U32,uint32_t). `RS` can be either 'strict' or 'relaxed'.

```
type xlupc_atomicX_read_RS(shared void *ptr);
void xlupc_atomicX_set_RS(shared void *ptr,type val);
type xlupc_atomicX_swap_RS(shared void *ptr,type val);
type xlupc_atomicX_cswap_RS(shared void *ptr,type oldval,
                                        type newval );
type xlupc_atomicX_fetchadd_RS(shared void *ptr,type op);
type xlupc_atomicX_fetchand_RS(shared void *ptr,type op);
type xlupc_atomicX_fetchor_RS(shared void *ptr,type op);
type xlupc_atomicX_fetchxor_RS(shared void *ptr,type op);
type xlupc_atomicX_fetchnot_RS(shared void *ptr);
```

Listing 2.7: XL UPC Atomics Interface.

Accelerated Collectives

UPC collectives are implemented completely within the runtime system. The compiler redirects UPC calls to runtime entries with only some minimal analysis in certain situations. In turn, the runtime transforms UPC pointers-to-shared to regular pointers and calls appropriate collectives within PAMI.

Other Runtime Optimizations

On a platform that supports simultaneous multithreading, thread binding turns out to be crucial. The runtime system provides flexible control on how to bind UPC threads to hardware threads. By default, no binding is performed while a specific mapping is performed with proper command line arguments. The auto-binding option is recommended. Huge memory pages usage is another runtime option, which instructs the memory allocator to use huge memory pages.

Chapter 3

Experimental Setup

This chapter explores characteristics of hardware environment, the benchmarks used in the evaluation, and finally presents a performance characterization of two benchmarks. The hardware description provides information about the hardware in a single node and presents the interconnect network. Furthermore, this section presents an evaluation of some key performance characteristics, such as latency, all-to-all and point-to-point bandwidth. The benchmark section introduces the available microbenchmarks and benchmarks used later in this work. The final part of this chapter presents execution time breakdowns of two applications to identify the performance bottlenecks.

3.1 The hardware environment

This dissertation uses the POWER7-IH (or Power 775) scalable High Performance Computing platform [63] for the evaluation. This platform consists of a number of Power7 [64, 65] nodes, connected using the Power7 Hub interconnect [66]. The entire system, including the processors, memories, and interconnect, is water cooled. The water dissipates the thermal waste by circulating components over the system and it is exchanged with the facility's chilled water.

Each node has four chips which constitute a 32-core Symmetric Multiprocessor (SMP) node. Chips are directly connected to each other at a peak of 48 GB/s. The Power7 processor has 32 KBytes instruction and 32 KBytes L1 data cache per core, 256 KBytes 2nd level per core, and a 32 MByte 3rd level cache shared per chip. Each core is equipped with four SMT threads and 12 execution units. The peak performance of each P7-IH chip is 246 GFlops/s resulting in a peak of 984 GFlops/s on each node. The Power7 processor implements efficient adaptive power management techniques compared to predecessors of PowerPC CPU family [67]. The microarchitecture implements a set of integer and floating point SIMD instructions, referred to as Vector Scalar eXtensions. Finally, each chip has two memory controllers supporting a total of eight DDR-3 memory channels. The size of available main memory is 128 GBytes per node.

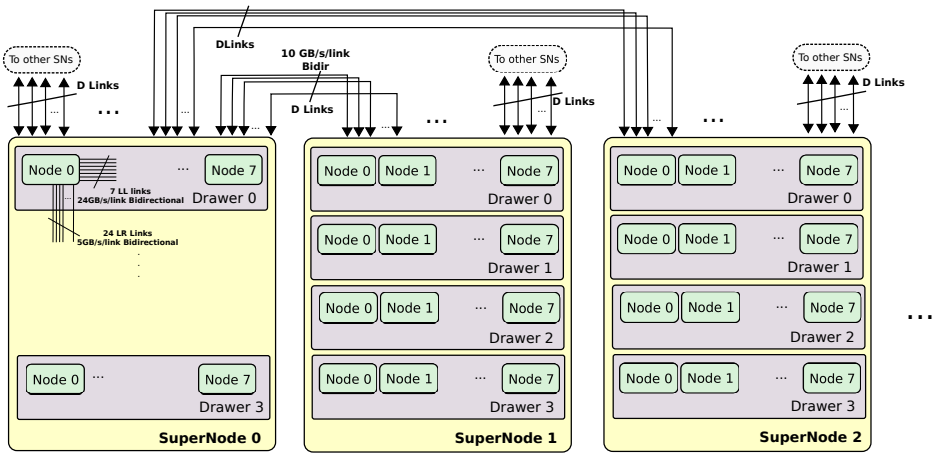

Figure 3.1: Architecture of the machine.

The machine is organized in drawers. Each drawer consists of eight nodes, adding up one TByte of memory and 7,87 Tflop/s. Four drawers are connected to create a supernode (1024 cores, 31.4flops/s). More supernodes can be added to increase the capacity of the system. A critical part of the Power 775 system is the custom interconnect chip that connects modules and expands the scalability of the system. High network performance is achieved by using the Power7 Hub interconnect or Torrent Chip. The Hub chip is connected with the four Power7 chips through four links, of 24GB/s each. The Hub chip contains seven links for communication between different nodes on the same drawer (LL links), 24 links for intra-supernode communication (LR links), and 16 links for inter-supernode communication (D links). Links between nodes in the drawer are electrical and intra/inter-supernode links are optical. Local drawer links have a combined bandwidth of 336 GB/s, the inter-supernode 240 GB/s, and the inter-supernodes 320 GB/s. There is no additional communication hardware present in the system. Figure 3.1 presents the architecture of the machine.

The Hub chip can achieve a bandwidth of 19GB/s per link and each node is capable of injecting up to 50GB/s (unidirectional) or 32GB/s (bidirectional) into the network [68]. Communication patterns that use single point-to-point messages between nodes (one core) do not saturate any network link, but that aggregate messaging from a single node (32 cores) to another can be bottlenecked by the use of a single LL link. Figure 3.1 presents the bandwidth results for point to point communication between nodes using the UPC language. The machine represents a very effective architecture for High Performance Computing, with high single-thread performance combined with a rich interconnection network, it provides high performance for a variety of workloads [69]. A full Power 775 system may contain up to 512 super nodes (524288 cores) with a peak performance of 16 Pflops/s. The full system used for this dissertation consists of 56 supernodes (SNs), totaling 1742 nodes. Thus, two supernodes are connected using only eight D links.

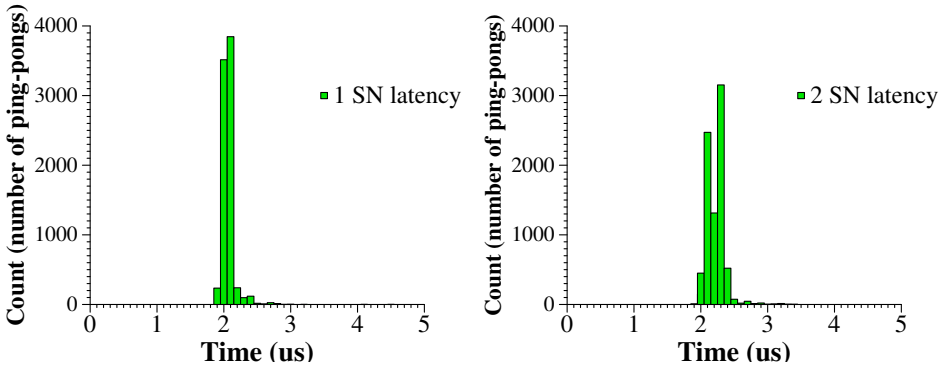

Figure 3.2: UPC benchmark ping-pong for one (left) and two (right) supernodes.

One important feature of the Power 775 architecture is the availability of multiple modes of data transfer, including optimized short message transfers (SHORT), arbitrary size messages (FIFO), and Remote Direct Memory Access (RDMA). SHORT data transfers are used for message sizes up to 128 bytes. FIFO data transfers are used for messages larger than 128 bytes. In the `RDMA` mode a process can read or write data on a remote compute node without involving its CPU. On top of this hardware, PAMI provides short active messages, active messages with large data transfers, and RDMA transfers. While the user doesn't need to know the effective remote address for active messages, active message performance is often slower than RDMA due to CPU involvement on the remote side. In RDMA mode, the data transfer is fast, but it requires that the thread which initiates the transfer knows the remote memory address where data will be read or written.

Interconnect and Routing

The Hub Chip's links are classified into two categories L, and D that allow the system to be organized into a two-level direct-connect topology, known as Dragonfly topology [70]. The D links are used between supernodes and there is at least one D link between two supernodes. The L links are used inside the supernode and they are sufficient for point-to-point communication. The topology forces the routes to be made up of small numbers of direct hops. Inside a supernode, any compute node communicate switch any other compute node using a single point to point L link. Across supernodes, a node inside the supernode that has direct point to point link with the remote supernode, uses one L hop to get to the destination node at the most. Finally, at the destination supernode, only one L hop is sufficient to reach the final destination. Thus, the maximum path for direct connections is L-D-L.

The system is capable of providing low latency communication down to *2*

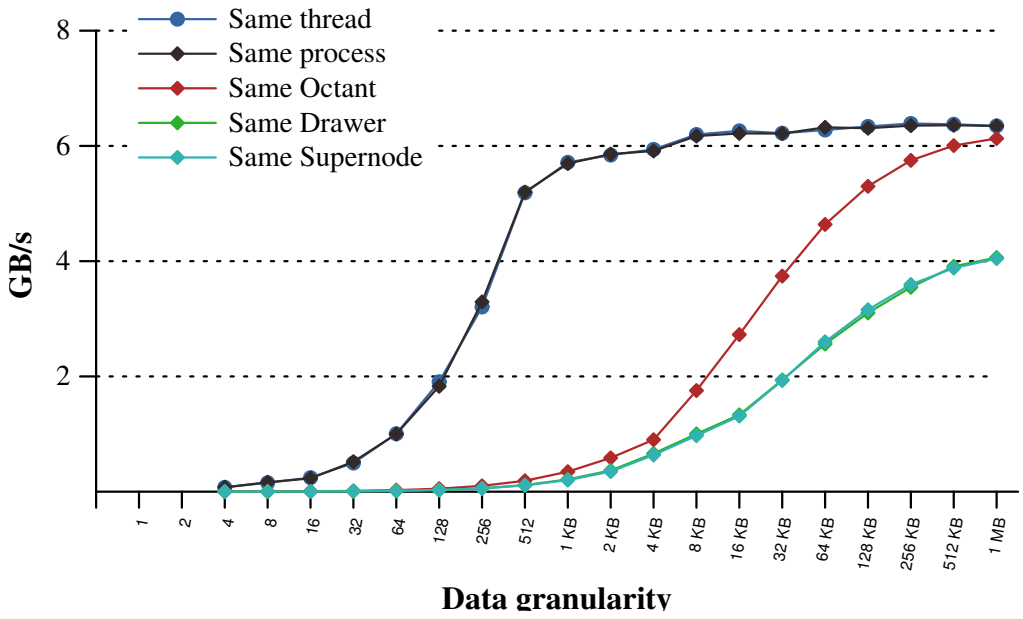

Figure 3.3: Unidirectional point-to-point bandwidth between different two cores.

Scenario	Links	Max Links (GB/s)	Max Hub Chip (GB/s)	Measured (GB/s)	Limited
2 Nodes	1 LL	96	74.36	61.5	H Chip
1 Drawer	16 LL	1536	297.44	287.5	H Chip
1 supernode	256 LR	5120	1189.7	1046.25	H Chip
2 supernodes	8 DL	320	2379.5	187.5	Links

Table 3.1: Theoretical peak uniform all-to-all bandwidth versus measured bandwidth for different machine configurations.

us intra-supernode and *2.3 us* for inter-supernode communication. Figure 3.2 presents the round trip latency between two different nodes. Indirect routing between two nodes in different supernodes increases the latency. Table 3.1 presents the uniform all-to-all bandwidth using an UPC benchmark. Notice the decrease of the available bandwidth between one and two supernodes. The inter-supernode links are likely to become the performance limit for various applications that require large number of nodes. Figure 3.3 shows the bandwidth achieved between two UPC threads in different scenarios. Note that each core achieves less than the size theoretical peak of the link.

Collectives Support

The Hub Chip provides specialized hardware to accelerate collective operations: the Collectives Acceleration Unit (CAU) [66]. The CAU unit integrated in the IBM Power 775 Hub provides offload and acceleration for broadcast and data reduction up to 64 bytes of data. For reduction, it supports NO-OP, SUM, MIN, MAX, AND, OR and XOR operations with 32-bit/64-bit signed/unsigned fixed-point operands or single/double precision floating-point operands. The benefits of deploying the CAUs are most evident in low latency operations. In the case of operations that require large bandwidth, such as broadcast or reduction on large data, the CAU unit may not perform better than the point to point versions.

UPC collectives are implemented entirely within the runtime system [55]. The compiler redirects UPC calls to runtime entries with only some minimal analysis in certain situations. The compiler transforms UPC pointers-to-shared to regular pointers and calls appropriate collectives within PAMI. Reduction and barrier collectives in XL UPC exploit the Collective Accelerated Unit (CAU) available on Power 775 when low latency performance is crucial.

3.2 Available Benchmarks

This work uses a number of benchmarks to evaluate the effectiveness of the optimizations. The benchmarks are written by other researchers and represent a variety of UPC programs that an UPC compiler can encounter in the real world. Table 3.3 summarizes the benchmarks used in this dissertation. The column type presents the computation type of the benchmarks, Lines of Code in terms of UPC source code for each benchmark, and the granularity of the messages. Note that most of the benchmarks contain communication with messages from one to eight bytes. The NAS FT and the bucketsort benchmark contain coarse-grained communication and they are used in the last part of the thesis.

3.2.1 Micro Benchmarks

This work uses microbenchmanks to demonstrate the effectiveness of the code transformations, calculate the upper limit of performance gain, and identify possible bottlenecks. Two different categories of microbenchmarks are used. The first category contains fine-grained communication. The evaluation uses these benchmarks to measure the potential performance benefit of the inspector-executor optimization. The second category is a group of four microbenchmarks that contain all-to-all communication patterns. The evaluation uses the second group of benchmarks to first identify potential network overheads and measure the possible benefits from the optimization.

The first category of microbenchmarks is based in a loop that accesses a shared array of structures. There are two variations of the loop: (i) The loop accesses consecutive array elements (stream like) and all four elements of the structure are

accessed; (ii) The loop accesses random elements of the array and all four structure fields are accessed. Listing 3.1 presents the first microbenchmark variation.

The second category of microbenchmarks emulates the all-to-all communication access shared array. There are four variations of this microbenchmark depending of the granularity of shared accesses. In all versions, each UPC thread executes the same code in the loop. In the **upc_memput** microbenchmark, the loop contains coarse-grained `upc_memput` calls. The **fine-grained get** contains shared reads and the **fine-grained put** contains shared writes.

```
1  typedef struct data{ double var0; double var1;
2                       double var2; double var3;
3  } data_t;
4
5  #define SIZE (1<<31)
6  shared data_t Table[SIZE];
7
8  double bench_stream_4_fields(){
9    uint64_t i;
10   double result0 = 0.0, result1 = 0.0;
11
12   for (i=MYTHREAD; i<SIZE-1; i+=THREADS){
13    result0 = Table[i+1].var0 + Table[i+1].var1;
14    result1 = Table[i+1].var2 + Table[i+1].var3;
15   }
16   return result0 + result1;
17 }
```

Listing 3.1: Microbenchmark kernel that reads four structure fields from a shared array.

3.2.2 Applications

Gravitational fish: The benchmark is a N-Body gravity simulation, using partial ordinary differential equations [49], using Euler's method. All the objects have the same mass. The benchmark places the fishes evenly along a cycle. There are three loops in the benchmark that access shared data, one for the computation of acceleration between fishes, one for data reduction, and one responsible for the calculation for the new position of fishes. The benchmark calculates the force for each object using all other objects using this equation :

$$F_i = G \times m_i \times \sum_{i=1}^{N} \frac{dx \times m_j}{(dx^2 + dy^2)^{3/2}}$$

The performance bottleneck of the baseline version is the first loop, the computation of the acceleration between fishes because it requires two shared accesses. The final loop, which calculates the new position of the fish, has complex upper

```
1  typedef struct { uint8_t r[IMGSZ]; } RowOfBytes;
2  shared RowOfBytes orig[IMGSZ];
3  shared RowOfBytes edge[IMGSZ];
4
5  void Sobel_upc(void){
6   int i,j,d1,d2; double magn;
7   upc_forall(i=1; i<IMGSZ-1; i++; &edge[i].r[0]){
8    for (j=1; j<IMGSZ-1; j++){
9      d1 = ((int) orig[i-1].r[j+1] - orig[i-1].r[j-1]);
10    d1 += ((int) orig[ i ].r[j+1] - orig[ i ].r[j-1])<<1;
11    d1 += ((int) orig[i+1].r[j+1] - orig[i+1].r[j-1]);
12
13     d2 = ((int) orig[i-1].r[j-1] - orig[i+1].r[j-1]);
14    d2 += ((int) orig[i-1].r[ j ] - orig[i+1].r[ j ])<<1;
15    d2 += ((int) orig[i-1].r[j+1] - orig[i+1].r[j+1]);
16
17    magn=sqrt((double)(d1*d1+d2*d2));
18    edge[i].r[j] = (uint8_t) (magn>255) ? 255:magn;
19   }
20  }
21 }
```

Listing 3.2: Sobel: UPC version of the kernel.

bound expression and cannot be normalized. The application overhead has been analyzed in the previous chapter in section 2.3.

Sobel: The Sobel benchmark computes an approximation of the gradient of the image intensity function, performing a nine-point stencil operation. At each point in the image, the result of the Sobel operator is stored on a second shared array. In the UPC variation [71], the image is represented as a two-dimensional shared array and the outer loop is a parallel *upc_forall* loop using a pointer-to-shared affinity test. The parallel implementation of the Sobel operator contains both local and remote operations. Listing 3.2 presents the kernels of the Sobel benchmark.

Mcop: The benchmark solves the matrix chain multiplication problem [72]. Matrix chain multiplication is an optimization problem that given a sequence of matrices, the problem finds the most efficient way to multiply these matrices together. The problem is not to actually perform the multiplications, but rather to decide the order in which the multiplications should be performed. The matrix data is distributed along columns, and communication occurs in the form of accesses to elements on the same row and column.

Fish-Predator: The benchmark simulates the evolution of predators and preys in an ocean over time [73]. The ocean is represented by a 2D matrix where each cell can either be empty or contain an individual predator (shark) or prey (fish). In each time step predators and preys can, after a certain time period, move

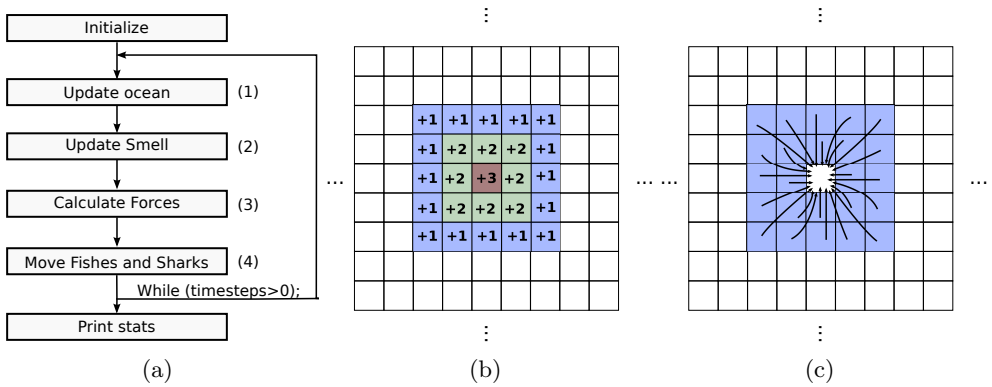

Figure 3.4: WaTor benchmark: architecture (left), smell update (middle), and force calculation (right).

or replicate themselves, to closer cells. The replication of preys and predators occur randomly. Figure 3.4(a) presents the architecture of the application. The benchmark uses four discrete steps to simulate the movement.

The collection of fish and sharks are in a doubly linked list that record their position, velocity, and age. The fish and shark movements are determined by forces that are calculated at each point of the two-dimensional grid that represents the ocean. Each cell of the grid stores the number of fishes and sharks. Furthermore, sharks have an additional field, the hunger level. Sharks need to eat or they will starve to death after a number of timestep.

The first step is the update of all fishes and sharks counters stored in each cells of the ocean (1). The next step involves the calculation of the smell of each cell on the grid (2). The FishSmell and SharkSmell are the weighted sum of the number of fish and sharks in a neighborhood of the grid point. At each time step the application uses the "smell" to compute a "force" on fish and sharks at that specific grid point. The fishes and sharks use the force to change their velocity. Each fish or shark contributes a smell of +3 to its cell, a smell of +2 to the cells with one cell distance and +1 to the grid with the distance of two cells. Figure 3.4(b) presents the update of the neighboring cells of the grid.

To compute the force vectors, the application searches at the 5×5 neighborhood grid points to find the fish and shark smell. Fish get a negative force from the shark smell or a random force if there are no sharks. Sharks get a force towards the fish smell if no fish smell is present (3). Figure 3.4(c) presents the calculation of the force between objects. Final step of the simulation (4) is the interaction between the fishes and the sharks. Sharks eat fishes if they share a grid point and they satisfy their hunger. If the fishes and sharks survive, then the application uses the forces to determine their new velocities and positions. Moving the fishes or the sharks means updating this pointer to the grid. Furthermore, the application also spawns fishes and sharks in this step.

```
1  #define TABSIZE (1L << LTABSIZE)
2  #define VLEN 512
3  #define NUPDATE (4L * TABSIZE)
4
5  shared [*] uint64 table[TABSIZE];
6
7  void calc(){
8    uint64_t ran[VLEN]; /* Random numbers */
9    uint64_t i, size = NUPDATE / THREADS;
10
11   for (i=0; i<size/VLEN; i++){
12       for (j=0; j<VLEN; j++) {
13         ran[j] = (ran[j] << 1) ^ ((int64) ran[j] < 0 ? POLY : 0);
14       }
15       for (j=0; j<VLEN; j++){
16         t1[j] = table[ ran[j] & (tabsize-1) ];
17       }
18       for (j=0; j<VLEN; j++) {
19           t1[j] ^= ran[j];
20       }
21       for (j=0; j<VLEN; j++){
22           table[ ran[j] & (tabsize-1)] = t1[j];
23       }
24   }
25 }
```

Listing 3.3: Guppie kernel in UPC.

Guppie: The guppie benchmark performs random read/modify/write accesses to a large distributed array. The benchmark uses a temporary buffer to fetch the data, modify them, and write them back. The typical size of this buffer is 512 elements. The selected size of data is static and evenly distributed among different UPC threads. The manual code modifications allow the compiler to optimize the shared accesses using the remote update optimization. In the UPC hand optimized version, the number of elements is set to one. Thus, the compiler collapses the loops and applies the remote update optimization [54]. Listing 3.3 presents the kernel code of the Guppie benchmark. The MPI version [74] 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 processor the local process performs the updates.

Bucketsort: The benchmark perform a bucket sort [75, 76] on 16 bytes length records. Each node generates its share of the records. Each thread uses a $\frac{17}{16} \times 2GB$ buffer in order to hold records received from other threads. These threads are destined to be sorted on this thread. Once this thread has generated all its share of records, it distributes the remainder of each bucket to the corresponding thread. Once this thread has received all appropriate data from each of the other threads,

it performs a sort on the local data. The benchmark contains coarse-grained accesses.

FT: The benchmark is part of the NPB NAS [77] suite. The benchmark solves a three-dimensional partial differential equation (PDE) using the Fast Fourier Transform (FFT). The benchmark creates an all-to-all communication pattern over the network. The benchmark contains coarse-grained accesses for communication between the UPC threads.

Benchmark	Type	Lines of Code (kernel)
Streaming read	Streaming read from next UPC threads Random updates	13
Random read	Random reads across all the UPC threads	14
Fine-grained get	All-to-all fine-grained reads	10
Fine-grained put	All-to-all fine-grained writes	10
upc_memget	All-to-all coarse-grained reads	9
upc_memput	All-to-all coarse-grained writes	9

Table 3.2: Overview of micro-benchmarks.

Benchmark	Type	Communication	Granularity	Lines of Code
Sobel	Sobel edge detection	Stencil	1 Byte	160
Fish Grav	N-Body gravity simulation using fishes	All-to-all	8 Bytes	246
Guppie	Random read/modify/write accesses to a large array	Random	8 Bytes	176
Mcop	Matrix chain multiplication problem	All-to-All / Irregular	4 Bytes	171
WaTor	Fish-Predator simulator	Stencil / Random	4 Bytes	792
Bucketsort	Integer sorting	All-to-All	16 Bytes >	270
NAS FT	FFT transformation	All-to-All	32 MBytes >	1307

Table 3.3: Overview of available benchmarks. We provide the source code line number for reference.

Chapter 4

Dynamic Data Aggregation

This chapter proposes a strategy based on the inspector-executor technique which aims to solve the problem of low efficient fine-grained accesses on shared variables. The goal of the optimization is to decrease the overhead created from the fine-grained accesses. The solution is to aggregate the shared references at runtime with the help of compiler transformation, when physical data mapping is missing. The optimization uses an improved inspector-executor loop transformation with the assistance of the runtime.

The inspector-executor transformation is presented in Section 4.1 as a possible solution to address the problem of fine-grained communication. However, the naive transformation introduces new problems to be addressed: (i) communication is blocking (ii) memory consumption with large number of loop iterations. For this reason, this section proposes to strip mine the loop (apply loop blocking) to address these two issues. Furthermore, the compiler versions the loops to avoid the runtime instrumentation overhead when the application runs with one process. Next, Section 4.2 presents an explanation of the implementation. First, it explains the compiler side implementation and the runtime support. It also presents how the implementation resolves the data dependencies and how the profitability is calculated. The final part of this chapter presents an extensive evaluation of the transformation using microbenchmarks and benchmarks (Section 4.3). The evaluation presents results of the microbenchmarks in terms of speedup, absolute performance numbers, and number of aggregated messages. Moreover, it presents an overview about the cost of the optimization. Finally, Section 4.4 discuss and concludes this chapter.

4.1 Approaches and solutions

The idea is to collect the shared addresses that are accessed in a loop, analyze, coalesce, and fetch them ahead of time before the data are required. Thus, the knowledge of the shared references before the execution of the loop is necessary in order to enable the use of an inspector-executor strategy [24, 23, 37, 26]. The idea

behind this is to collect the shared addresses that we want to access on the inspector loop, analyze them, and fetch them before entering the main loop. The executor loop reads the data from local buffers and makes the computation. Listing 4.1 presents the initial source code and Listing 4.2 illustrates the final form.

```
1 shared double B[N], A[N], C[N];
2
3 ...
4 // two remote reads
5 for( i=MYTHREAD; i<N-1; i+=THREADS)
6    A[i] = B[i+1]*C[i+1];
7 ...
```

Listing 4.1: Initial source code.

4.1.1 Inspector-executor strategy

The inspector-executor approach has two disadvantages: (i) the *pause issue*: the execution of the actual program is paused to analyze shared accesses and to fetch corresponding data. This pause creates artificial delays and unnecessary overhead on the execution of the program; (ii) the *resource issue*: the number of iterations of the loop may be high enough that leads the memory requirements to unacceptable levels. One possible solution to this latter issue is to inspect elements for a limited number of iterations. However, the performance gain in loops with large number of iterations will be proportional to the fraction of the loop iterations.

```
1  shared double B[N], A[N], C[N];
2   ...
3   /* Inspector loop: describe remote accesses */
4   /* to the runtime system */
5   for(i=0; i<N; i++){
6      __xlupc_add_access(&B[i+1]);
7      __xlupc_add_access(&C[i+1]);
8   }
9
10  schedule( ); /* Access analysis & prefetching */
11
12  /* Executor loop: retrieve prefetch data */
13  for( i=0; i<N; i++ ){
14     buff1 =__xlupc_sched_deref(&idx1,&B[i+1]);
15     buff2 =__xlupc_sched_deref(&idx2,&C[i+1]);
16     A[i] = buff1[idx1]*buff2[idx2];
17  }
18  __prefetch_reset();
19  ...
```

Listing 4.2: A simple inspector-executor approach

4.1.2 Double buffering

In order to overcome the drawbacks of the simple loop inspector, the compiler strip mines (or blocks) the main loop. The loop's iteration space is partitioned into smaller chunks or blocks. The inner loop is replicated and transformed into two new loops: inspector and executor loops. The chunk size is the number of iterations that will be collected and analyzed on each prologue execution, therefore we call it the `prefetch factor (PF)`. The runtime chooses the best PF to maximize the benefit without exhausting the resources, solving the *resource issue*.

```
1  shared double B[N], A[N], C[N];
2
3  ...
4  /* Prologue loop: inspect the 1st block */
5  for(i=0; i<PF; i++) {
6     __xlupc_add_access(&B[i+1]);
7     __xlupc_add_access(&C[i+1]);
8  }
9  __schedule(); /* Schedule the accesses */
10
11 for(j=0;j<(N-N%PF);j+=PF){
12    /* Inner prologue: inspect next blk */
13    for(i=j+PF; i<MIN( N, j+2*PF); i++){
14       __xlupc_add_access(&B[i+1]);
15       __xlupc_add_access(&C[i+1]);
16    }
17    __schedule(...); /* Schedule the accesses */
18    for( i=0; (i<PF); i++ ){ /* Inner main loop */
19       buff1 =__xlupc_sched_deref(&idx1, &B[i+1]);
20       buff2 =__xlupc_sched_deref(&idx2, &C[i+1]);
21       A[i] = buff1[idx1]*buff2[idx2];
22    }
23    /* Reset internal buffers/counters */
24    __prefetch_reset();
25 }
26 __prefetch_wait(); /* Wait for the last iteration */
27
28 for(i=N-N%PF; i<N; i++){ /* Epilogue/Residue loop */
29    buff1 =__xlupc_sched_deref(&idx1, &B[i+1]);
30    buff2 =__xlupc_sched_deref(&idx2, &C[i+1]);
31    A[i] = buff1[idx1]*buff2[idx2];
32 }
33 __prefetch_reset();
34 ...
```

Listing 4.3: A more advanced variation of inspector-executor with loop strip mining and double buffering technique.

To address the *pause issue*, the inspector loops are shifted one iteration block to inspect the next block of iterations, thus creating a pipelining effect. The

pipelining effect exploits the overlapping of block inspection/execution with the transfer block using a double buffering technique. The inspector loop collects the elements for the $(i + 1)_{th}$ block of iterations while the executor loop reads the coalesced data from a local buffer of the i_{th} block of iterations. The double buffering allows the execution of the i_{th} block of iterations during the transfer of the $(i+1)_{th}$ block. The shared data are prefetched ahead of time and the main loop executes the i_{th} iteration without having to wait because the i_{th} block of data is ready in the local buffers. Finally, the blocking loop transformation automatically creates a residual loop (epilogue) that executes the final iterations of the main loop. Listing 4.3 presents the transformed loop structure.

4.1.3 Loop versioning

In the final transformation versions the loop creates two variants: the transformed and the unmodified. The runtime is responsible for deciding whether it is profitable to run the optimized version or not, and if so, decide on the best number of iterations to prefetch (prefetch factor). The prefetch factor (PF) is the value that is responsible for the branch condition. It is the result of runtime call and is also used as blocking factor for the strip mined loop. For that purpose, the runtime takes into account the number of shared accesses per iteration, the number of iterations and the number or processes. Section 4.2.2 provides the implementation details. For instance, in the case where there is only one node in the system and all UPC threads are located in the same node, the benefits from the optimization are very limited and not enough to compensate the overhead. Consequently, the runtime will decide not to apply the optimization. Figure 4.1 presents the final form of the transformed loop structure.

4.2 Implementation

This section presents a detailed explanation of the implementation of the optimization proposed. First the compiler transformation is presented and then the required runtime support. In short, the compiler applies the loop transformations and inserts the proper calls between different parts of the loop structure. The runtime is responsible for profitability analysis, for keeping the list of shared references, for analyzing and coalescing them, and for retrieving the data from local buffers.

4.2.1 Transformation algorithm

Algorithm 1 describes the steps of the compiler algorithm. First, the compiler checks the memory architecture and the consistency model. The architecture must be either distributed memory or unknown during the compilation time and the memory model must be relaxed. The compiler applies the transformation in four phases: in the first phase, it does the gathering of loop candidates and the

PrefetchAndCoalesceSharedRefs(*Procedure p*)

1: **for each** candidate loop structure L_i in p **do**
2: $RefList \leftarrow \emptyset$;
Phase 1 - Gather Candidates
3: **for all** Shared mem reference **and** is non-local R_s in L_i **do**
4: $RefList$.Add(R_s);
5: **end for**
6: **if** $RefList$ is $\emptyset$ **then**
7: continue;
8: **end if**
Phase 2 - Replicate and create loops
9: $branchExpr \leftarrow prefetch\_factor\_call$
10: $\{L_i, nativeLoop_i\} \leftarrow Version(L_i, branchExpr)$
11: $prolog_i \leftarrow Replicate(\ L_i\ )$
12: $\{residualL_i, innerloop_i\} \leftarrow StripMine(L_i)$
13: $innerprolog_i \leftarrow Replicate(\ innerloop_i\ )$
Phase 3 -Modify inspector loops
14: $Preserve\_expr(prolog_i, innerprolog_i)$;
15: $Preserve\_vars(prolog_i, innerprolog_i)$;
Phase 4 - Insertion of runtime calls
16: $(prolog_i^{BODYEND}, innerprolog_i^{BODYEND}).Add($ schedule)
17: $L_i^{BODYEND}.Add($ schedule_wait)
18: $(innerloop_i^{BODYEND}, residualL_i^{BODYEND}).Add($ sched_reset) ;
19: **for each** shared mem ref R_s in $RefList$ **do**
20: $stmt_s \leftarrow SHARED\_STATEMENT(\ R_S\ )$
21: **for each** statement $stmt$ in $prolog$ **do**
22: **if** $stmt = stmt_s$ **then**
23: $prolog_i^{stmt}.Add($add_access_call)
24: $prolog_i^{stmt}.Remove()$
25: **break**;
26: **end if**
27: **end for**
28: **for each** statement $stmt$ in $innerprolog$ **do**
29: **if** $stmt = stmt_s$ **then**
30: $innerprolog_i^{stmt}.Add($add_access)
31: $innerprolog_i^{stmt}.Remove()$
32: **break**;
33: **end if**
34: **end for**
35: **end for**
36: **for each** shared mem ref R_s in $RefList$ **do**
37: $stmt_s \leftarrow SHARED\_STATEMENT(\ R_S\ )$
38: **for each** statement $stmt$ in $innerloop_i$ **do**
39: **if** $stmt = stmt_s$ **then**
40: $innerloop_i^{stmt}.Add($ buf=deref)
41: $innerloop_i^{stmt}.Replace(stmt_s^{expr}, buf)$
42: **end if**
43: **end for**
44: **for each** statement $stmt$ in $residualL_i$ **do**
45: **if** $stmt = stmt_s$ **then**
46: $residualL_i^{stmt}.Add($buf=deref)
47: $residualL_i^{stmt}.Replace(stmt_s^{expr}, buf)$
48: **end if**
49: **end for**
50: **end for**
51: **end for**

Algorithm 1: Inspector-executor compiler algorithm.

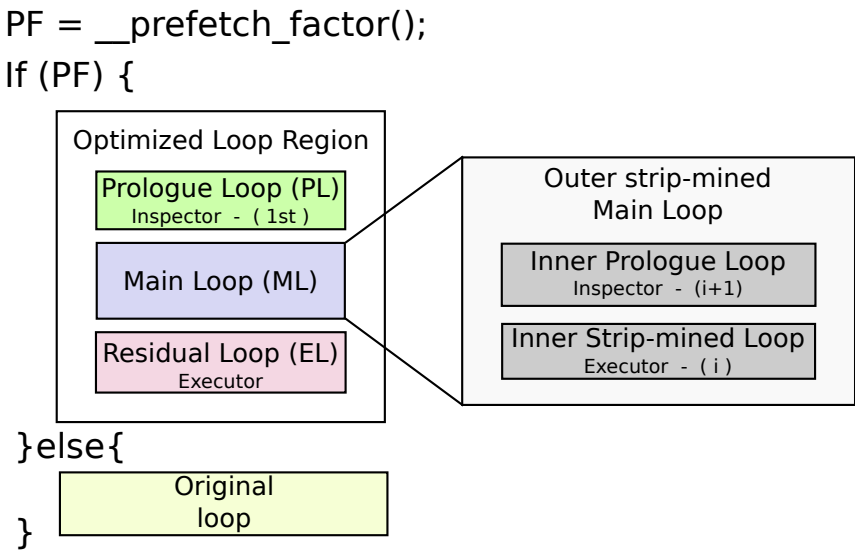

Figure 4.1: Final version after the loop versioning.

collection of shared references. The second phase applies the loop transformations. In the third phase, the compiler modifies the inspector loops to maintain only the expressions that are responsible for creating the indexing value and for preserving the values of the loop that is being modified. In the fourth and final phase, the compiler inserts the proper runtime calls to support the optimization.

Phase 1: Gathering of candidates

Initially, the algorithm collects candidate loops containing shared array references. The algorithm uses the following criteria to evaluate whether the loop is a candidate for prefetch analysis: (1) The loop must not contain any break/continue statements, (2) it must be normalized, and must not have any (3) lexicographic negative dependencies between iterations (line 1).

The first rule (1) ensures that the loop has a single entry and exit point. This ensures the simplicity of the transformation. Implementing a more complex version that supports these types of loops is possible. However, we have to take into account the out-of-order execution of the runtime calls. This requires additional checks at runtime that can decrease the application performance.

The second requirement, the normalized loop (2), is necessary to optimize loops. There are two reasons for this requirement. First the loop dependence analysis is simpler because the distance of dependency can be calculated easier than the non-normalized loops. The second reason is to avoid loops that modify data structures that are used for the exit condition. These loops are had to predict and require again changes on the runtime for arbitrary library call execution. Furthermore, this requirement ensures that the low/upper bound expression of the loop and the iteration step are not a procedure call. In our implementation,

the XL UPC compiler framework normalizes most of the loops before entering to the presented transformation.

The final requirement, the loop must not have lexicographic negative dependencies [78], ensures the good performance of the program. For example, assuming the shared arrays 'a', we can have this loop:

```
for ( i = 1; i<N; i++){
    a[i] = a[i-1];
}
```

The transformation prefetches the accesses `a[i-1]`. However, the program will read the wrong element on position `a[i]` on the next loop instance. Section 4.2.3 presents a solution for these types of loops, unfortunately with a drop in the application's performance.

Following this selection, the compiler further analyzes the candidate loops by inspecting the shared references they contain. The algorithm collects the shared references that are not local (line 3). If the collected list of prefetching elements is empty then the algorithm continues with the analysis of the next loop (line 6).

Phase 2: Loop transformations

The compiler versions the loop in two new variants in line 10: the unoptimized native version and the optimized version. In addition, it inserts a runtime call before the two versions of the loop. This call will returns at runtime the number of iterations to be prefetched: the prefetch factor (PF). This value is used as a profitability condition to select the proper loop version.

In the optimized version, the loop is transformed into two new loops, a prologue loop and a main loop (line 11). The main loop is strip mined, and its inner and outer loops are replicated (line 13) resulting in the inner prologue loop and the inner strip mined main loop (Figure 4.1). The prefetch factor (PF) is used as loop strip size. In the inner prologue loop, the bounds are modified to prefetch the next iteration of the inner main loop. The loop blocking transformation automatically create a residual loop (or epilogue loop) to execute the last remaining iterations, in case the PF doesn't divide exactly the upper bound of the loop. Finally, the algorithm examines the prologue loops to ensure that they do not contain procedure calls with side effects.

Phase 3: Inspector loop adjustment

Two challenges must be addressed before the insertion of runtime calls. The first challenge is to preserve the expressions that modify the index variables. In line 14, the algorithm uses control and data flow graph analysis [42] to search for expressions that modify the symbols used for the calculation of the index. The other challenge is to preserve the variables that the inspector loop modifies during the execution. In this case, we store the values to temporary variables before the body

of the loop and then we restore them after the execution of the inspector loops (line 15).

Phase 4: Runtime calls insertion

In the final step, the compiler inserts the following runtime calls:

- The *schedule* call is inserted before the main loop and between the start of the inner loop and the inner prologue (line 16). These calls instruct the runtime system to analyze, coalesce the accesses, and issue network communication. Furthermore, the schedule call blocks until the transfer of the previous data block is complete.
- A *schedule_wait* call is inserted after the end of the main loop and before the residual loop (line 17). If the PF does not divide the loop upper bound, then the program must wait for the last block of data to be transferred before the execution of the residual loop.
- The compiler inserts the *schedule_reset* calls after the end of executor loops: the inner loop and the residual loop (line 18) to recycle the internal runtime data structures for the next prefetching phase.
- The *add_access* call is inserted (lines 23, 30) in the body of inspector loops, for each shared reference. These calls describe the shared references to the runtime system. The compiler passes through the statements of the inspector loops, to replace the statements containing shared references with the runtime call.
- For each shared reference, a *dereference* call is inserted inside the executor loops: inner main loop and the residual loop (lines 40, 46). The dereference call returns a buffer and sets the proper value on a temporary variable for indexing the buffer. The compiler replaces each expression containing the shared reference with a local access by using the aforementioned buffer and index (lines 41, 47).

4.2.2 Runtime support

The optimization requires runtime support to perform prefetching and coalescing. The runtime is responsible for four tasks: (a) decides if the optimization 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. Thus, the proper selection of the algorithms and data structures have significant impact on the performance of the application.

The first task is to decide whether the loop transformation is beneficial and calculate prefetch factor. Algorithm 2 calculates the prefetch factor. The runtime checks if the number of processes is more than one and if the number of UPC

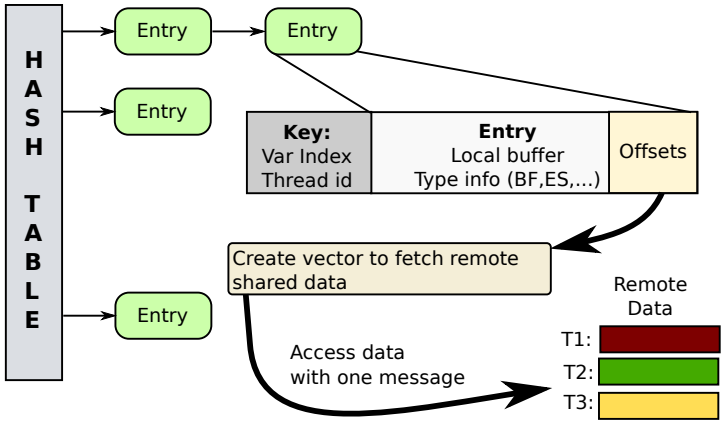

Figure 4.2: Runtime internal implementation.

threads is greater than the upper bound of the loop (line 1). If yes, then the runtime will decide to execute 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 using thread communication is less than the overhead of scheduling optimization, which requires keeping information about shared accesses and analyzing them. Furthermore, we put 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. We use the upper bound minus one to ensure that the program will be enter at least one time in the main loop. If the prefetch factor is less than two, then the unoptimized version is used.

The second task is to collect and store information about shared accesses in

Prefetch_factor(*int num_elems*, *int upper_bound*)

```
 1: if XLPGAS_NODES == 1 or upper_bound ≤ XLPGAS_NODES then
 2:     return 0;
 3: end if
 4: max_loop ← upper_bound − 1;
 5: PF ← MAX_FETCH/num_elems;
 6: if PF ≥ max_loop then
 7:     PF ← MAX;
 8: end if
 9: if PF ≤ 2 then
10:     PF ← 0;
11: end if
12: return PF;
```

Algorithm 2: Calculation of Prefetch Factor.

the inspector loops. For each shared access the runtime stores information about the shared variable, the related offset, the blocking factor (BF) and element size (ES) into a hash table. For each pair of variable and UPC thread identifier, the runtime searches the hash table. If an entry exists, then the runtime inserts the offset, otherwise the runtime creates a new entry and inserts it in the hash table. On each entry of the hash table, the runtime maintains an array of the offsets that the runtime collects. The runtime does not issue any communication request during the collection phase. Moreover, the runtime uses a memory pool and buffer recycling to avoid the dynamic memory allocation overhead. On each call of the 'reset' procedure, the runtime resets the internal data structures, including the hash entries.

The third task is the 'scheduling' of accesses. A runtime call is responsible for analyzing, coalescing and prefetching 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 faster. Second, removing duplicates decreases the transfer size in applications that have duplicates, such as stencil computation. Finally, the runtime prepares a vector of offsets for fetching data. The transport library uses one-sided communication to achieve high throughput with low overhead. Figure 4.2 presents the internal structure and the analysis technique of the runtime.

The fourth task involves data retrieval from the local buffers. The runtime call returns a pointer to the local buffer and stores the index variable with the proper value. Internally the runtime first tries to calculate the index value directly by using an auxiliary table. If this fails, searches the offset table using a binary search algorithm. One of the key optimizations of the runtime is the handling of shared data that belong to the same node. The runtime does not collect nor analyze the shared references if they are locally accessible. In the dereference call, the runtime simply returns a pointer to the local data.

4.2.3 Resolving Data Dependencies

```
1 #define N 8192
2
3 int compute(shared int *ptr1, shared int *ptr2){
4   int i;
5   for(i=0;i<N-1;i++){
6     ptr1[i] = ptr2[i];
7   }
8 }
```

Listing 4.4: Example of possible pointer aliasing.

In a parallel loop with loads and stores on shared objects, the compiler may not be able to determine the data dependencies at compile time and must assume

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, i);

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

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

Figure 4.3: The runtime resolves dependencies with the help of the compiler.

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 4.4 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 guarantee memory consistency, the compiler creates the shared write calls with an additional argument that notifies the runtime to make additional checks for outstanding transfers. The compiler will set this flag to true if there is an overlap between shared addresses or if the compiler fails to resolve the alias dependencies. The runtime handles the shared data stores in one of the three following ways:

- Compiler signals that there is no overlap with other shared objects that application modifies. In this case, the runtime does not execute any additional code.
- Program writes on shared data and there is a copy of the shared data on a local buffer. Runtime issues the remote store and updates the local buffer to maintain the consistency.
- There is overlap between the shared stores and the transferred data from a remote node. In this case, the runtime waits for the transfer to be completed and overwrites the prefetched data.

Figure 4.3 presents the user's code and the runtime implementation. If the compiler fails to determine the dependencies between variables, it sets the variable *update_local*.

4.3 Experimental Results

This section presents experimental results using two microbenchmarks and five benchmarks. First, the evaluation uses microbenchmarks to calculate an upper

limit on the performance gain and identify possible bottlenecks. Next, the performance of the benchmarks is presented. Moreover, this section provides an analysis of the overheads by presenting application time breakdowns. Finally, we briefly analyze the cost of the optimization in terms of code increase, compilation time, and runtime memory requirements.

4.3.1 Benchmark versions

The evaluation uses five different benchmark versions:

- The *Baseline* version, compiled with dynamic number of threads and the inspector-executor optimization disabled. The number of UPC optimizations that the compiler applies without the physical mapping knowledge is limited to privatization of some shared accesses inside the *upc_forall* loops.
- The *Aggregation* version, where the compiler applies the inspector-executor optimization that prefetches and coalesces shared references at runtime.
- The *hand-optimized* shows the performance of the benchmarks using coarse-grained communication and manual pointer privatization. The manual work focuses on strip mining the loops and the use of collective communication mechanisms when possible.
- The *MPI* version contains coarse-grained communication. It uses collective communication when possible and does not use the non-blocking mechanisms for fairness.

When the optimization is unable to find any opportunities for coalescing shared accesses, the algorithm returns without any modification on the program. In other cases, the compiler applies the optimization, but the number of iterations at execution time is low in order to benefit from the optimization. The runtime library will detect this incidence at runtime and the program executes the unoptimized version of the loop. In this case, the overhead incurred by calling the runtime for loop selection, is relatively small compared with the fine-grained accesses.

All runs use one process per UPC thread and schedule one UPC thread per Power7 core. The runtime prefetches at most 4096 iterations (*MAX_FETCH*). Each UPC thread communicates with other UPC threads through the network interface or interprocess communication. The UPC threads are grouped in blocks of 32 per node and each UPC thread is bound to its own core. The experimental evaluation runs each benchmark five times. The results presented in this evaluation are the average of the execution time for the five runs. In all experiments, the execution time variation is less than 5%. We run one iteration of the optimized loop before the actual measurement to warm-up the internal structures of the runtime. All benchmarks are compiled using the '`-qarch=pwr7 -qtune=pwr7 -O3 -qprefetch`' compiler flags.

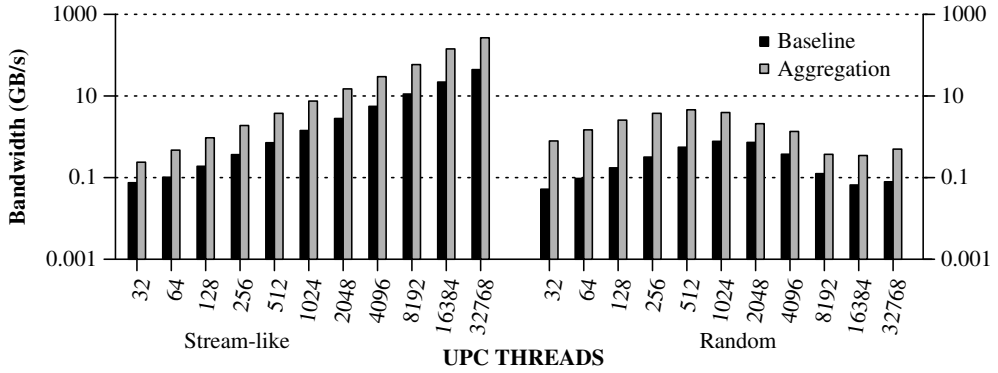

Figure 4.4: Performance in GB/s for the microbenchmark reading four fields from the same data structure in streaming and random fashion.

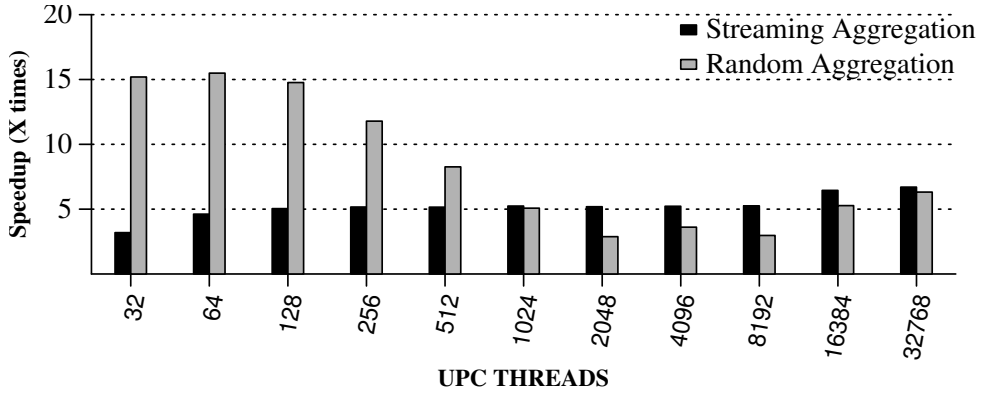

Figure 4.5: Achieved speedup for the two microbenchmark variations.

4.3.2 Microbencmarks Performance

Microbenchmanks are used to demonstrate the effectiveness of the code transformations and identify pathological defects. Figure 4.4 presents the achieved performance of the microbenchmark in GB/s, using different variations. The speedup of the inspector-executor version (*Aggregation*) varies between 3.1x and 6.7x over the baseline using the first variation of the microbenchmark (Figure 4.5). The first microbenchmark reads data from the neighboring UPC threads. Thus, internally the runtime creates one entry in the hash table and the memory overhead is limited. In this microbenchmark version, the runtime coalesces (or aggregates) on average 4096 elements in one message. Moreover, we observe a slight increase in the achieved speedup for more than 16384 UPC threads, most likely due to higher latency and network contention. On the other hand, when reading elements in a random way, the speedup varies from 3.2x up to 21.6x. In the case of random access pattern, we analyze three irregularities: (i) benchmark with random access pattern achieved better performance from the stream-like with 256 or less UPC

benchmarks; (ii) the performance gain (speedup) decreases in random access when the number of the UPC threads is increasing; (iii) the performance of random reads decreases for more than 1024 UPC threads.

Performance of random access versus stream-like

The first interesting result is that the random access achieves better bandwidth compared with the stream-like variation when the prefetching is enabled and we run the benchmarks with 256 or less UPC threads. For example, in the stream-like variation, the benchmark achieves 2.1 GB/s in contrast with the 4.5 GB/s in the random access. The reason for that is the Hub chip architecture. The Hub chip has 7 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 [79].

To investigate this behaviour we focus at the runs with 256 UPC threads. In the case of the stream-like, the runtime fetches $entries{*}sizeof(field) = 4K{*}8Bytes = 32KBytes$ packets from the neighbor UPC thread. The slowest part connects the nodes: when thread #31 that belongs to node 0 fetches from thread #32 that belongs to node 1. In this case, the unidirectional bandwidth will be around 2GB/s using 32 KByte packets. In the case of random access, we can calculate the number of entries in the hash table, assuming uniform random accesses:

$$Entries = \frac{\frac{MAX\_FETCH}{\#Elems\_loop}}{\#UPC\_THREADS} = \frac{\frac{4096}{4}}{256} = 4.$$

That means that we need to fetch $fields{*}entries{*}sizeof(field) = 4{*}4{*}8 = 128$ Bytes packets per remote thread. That means that we have one request of 128 Bytes per remote thread, and we are going to use all the 7 links with other nodes. We have 32 threads on the same node, so we will have 32 packets of 128 bytes per link. In the best case scenario, the packets will arrive at the same time and the hardware coalesces them creating one big packet of 4 KBytes. Packets of 4 KB have around 860 MB/s point-to-point unidirectional bandwidth. Thus, the aggregate bandwidth can be up to $7 * 0.86 = 6.02GB/s$ at best, which is much higher than the 2 GB/s of the streaming-like variation. Although, the hardware does not always coalesce the packets because they arrive at different time, when the achieved aggregated bandwidth of the interconnection network increases.

Overall, in the streaming benchmark only one of the threads in the node communicates with the neighbor node and therefore, only one of the Hub links is used, decreasing the bandwidth to the maximum that can be achieved. In contrast, the random pattern benchmark all nodes communicate, assuming an average of 4 remote accesses per thread. Thus, the traffic goes through the seven available links, achieving higher effective bandwidth than the stream-like version.

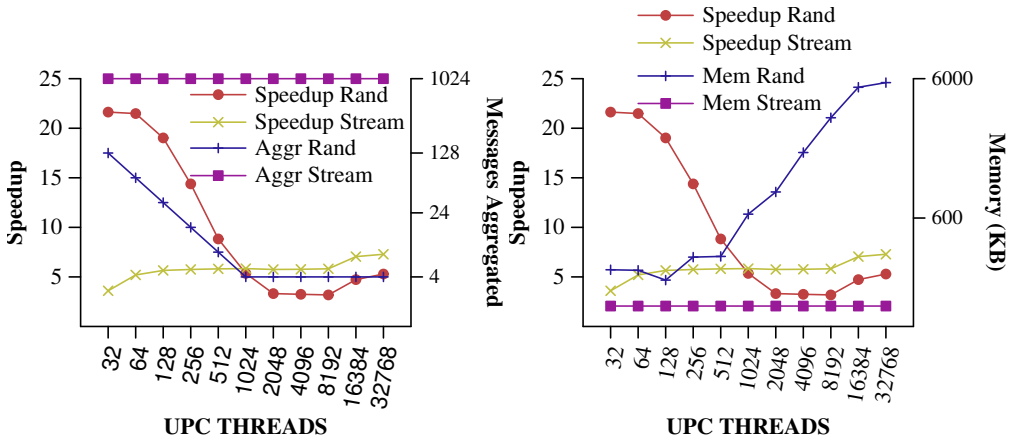

Figure 4.6: Achieved speedup for the two microbenchmark variations compared with the number of messages aggregated (left), and speedup compared with the memory consumption of the runtime (right).

Speedup decrease of random access

The second observation is that the performance gain (speedup) of the random access decreases, but in stream-like remains constant. There are two reasons for this outcome: (i) The memory consumption of the runtime system, and (ii) the number of coalesced messages. Figure 4.6(a) presents the number of coalesced messages per UPC thread compared with the number of threads running. We can see the correlation between the speedup and the aggregation of the messages. The problem is to be found in the iterations to inspect (prefetch factor). The maximum prefetch factor is 4096, thus for random accesses using 1024 UPC threads, the entries are: $\frac{4096/4}{1024} = 1$ access per thread, assuming uniform distribution. However, the benchmark reads four elements of the structure using the same index. Thus, the runtime coalesces four accesses in all cases. An additional factor that influences negatively the performance is the impact of the memory allocation. Figure 4.6(b) presents the memory consumption per UPC thread and the achieved speedup. We observe that the memory consumptions increases on each doubling of UPC threads. In this case, the runtime adds new entries in the hash table and allocates a buffer for each UPC thread. The speedup decreases from 5.3x to 3.0x. On the other hand, the memory consumption is constant in the stream-like benchmark and the runtime aggregates always 1024 messages.

Random Reads Performance Decrease

The final observation is that the performance of the random reads decreases for more than 1024 UPC threads, with or without prefetch. This decrease in performance 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) [63] can have three hops at most which consist of no more than two L hops and at most one D hop maximum. The advantage of this topology is that it improves the bisection bandwidth over other topologies such as fat-tree interconnects and eliminates the need for external switches. However, the architecture of the interconnection limits the performance in the case of the random accesses pattern when most of the traffic is routed through the remote links.

To prove that the interconnect is the bottleneck in the random access pattern, we calculate for each node, how much of traffic will use the D links each supernode for the communication. In all cases we assume uniform traffic distribution. In order to calculate this value, we first detract the number of threads that belong to the same supernode because there is direct connection between them. Secondly, we detract the number threads that belong to other threads.

First we calculate with how many threads, that reside on different supernodes, one thread can communicate using direct connection. In our setup, the cluster uses eight inter-supernode (8D) links to communicate together [80]. Thus, the number of links one supernode has with other supernodes is:

$$number\_of\_links = (\#supernodes - 1) \times 8$$

Because each supernode has 32 nodes, and each node contains one Hub chip. On average each node has:

$$remote\_links\_per\_node = \frac{(\#supernodes - 1) \times 8}{32}$$

In our experiments we use 32 UPC threads per node. The total number of UPC threads that use direct connection are:

$$remote\_links\_per\_thread = \frac{(\#supernodes - 1) \times 8 \times 32}{32} = (\#supernodes - 1) \times 8$$

Moreover, each node has 31 direct connections with other nodes inside the supernode. The total number of direct connections inside the thread are:

$$intra\_links = threads\_per\_node \times nodes\_per\_supernode = 32 \times 31 = 992$$

Each node contains 32 UPC thread. Thus, we assume that all communication between these threads are local ($intra\_node\_threads$). Combining all the equations together, the amount of traffic that is using links with more than one hop is:

$$\frac{threads - remote\_links\_per\_thread - intra\_links - intra\_node\_threads}{threads}$$

Table 4.1 presents the percentage of traffic that uses more than one hope. Overall, the interconnection of the supernodes burdens the performance in random access patterns. This is due to the saturation of the remote links in more than 1024 UPC threads.

Number of UPC threads	1-hop Links (%)	Links with > 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 4.1: Percentage of traffic that uses remote and local links.

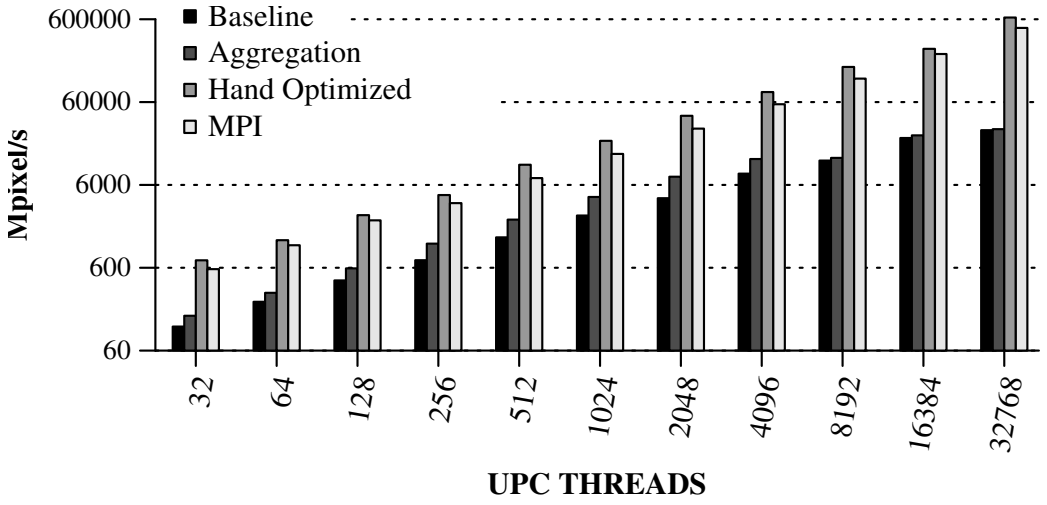

Figure 4.7: Performance numbers for the sobel benchmark using different versions.

4.3.3 Applications Performance

Sobel

The sobel benchmark communicates with neighboring UPC threads only at the beginning and at the end of the computation. The kernel of the benchmark has nine shared reads, and the compiler will create additional library calls in the inspector loops. Figure 4.7 illustrates the performance numbers for the sobel benchmark in mega-pixel/s[1]. The inspector-executor (*aggregation*) optimization achieves a performance gain between +15% and +60%. The relatively low performance gain compared with the microbenchmark and the gravitational fish benchmark, is due to the good shared data locality. For example, only 1.6% of the shared accesses are remote, running with 2048 UPC threads. The shared data has been distributed by blocks of rows to minimize remote accesses. One interesting characteristic of the sobel benchmark is that the runtime coalesced 258 packets into one remote message, independently of the number of the UPC threads. Thus, the number of memory consumption from the runtime is constant in all runs at 62 KBytes.

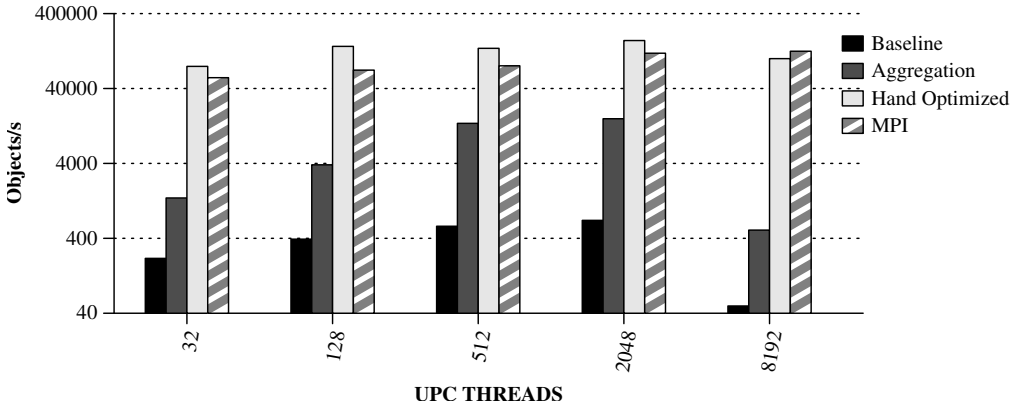

Figure 4.8: Performance numbers for the gravitational fish benchmark using different versions.

Finally, and as expected, the manually optimized variation has better performance than all other variations. The programmer is able to coalesce 65536 elements (one row) in one transfer in the manually optimized version. Thus, the programmer uses only local data inside the execution loop, avoiding the creation of runtime calls from the compiler.

Gravitational Fish

The Fish benchmark has very low performance due to fine-grained accesses (Figure 4.8). The inspector-executor (aggregation) optimization gives a speedup between 6.3X up to 23.6X compared to the baseline, thus proving the usefulness of the optimization. Furthermore, we observe that the performance drops significantly for more than 1024 UPC threads. Figure 4.8 includes numbers only up to 8192 UPC threads because runs with 16K or more threads are not practical. There are two sources that burden the performance of the application: (i) the limitations resulting from the architecture of the interconnect network and (ii) the way the data are stored and accessed. First, the benchmark saturates the inter-supernode links for the same reasons that the second microbenchmark (random access) has low performance for more than 1024 UPC threads, in a similar way fish saturates the inter-supernode links. Secondly, all UPC threads access data in a streaming fashion starting from the first UPC thread. Thus, for the first iterations of the loop, all the UPC threads will try to access the data on the first UPC thread at the same time. Moreover, the hand-optimized and MPI versions are not exhibit this pathogenic behaviour for two reasons. First, they use collective communication (broadcast and reduction) to accelerate the transfers. Second, they use coarse-grained communication that utilizes more efficient the network.

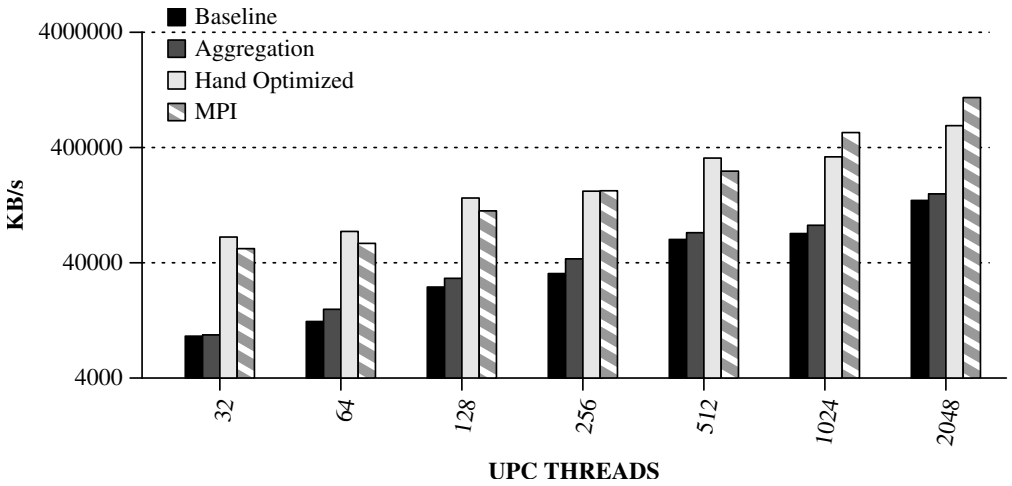

Figure 4.9: Performance numbers for the WaTor benchmark using different versions.

WaTor

The aggregation and the static coalescing give a speedup from 3.8x up to 15.6x compared with the baseline version in the WaTor benchmark (Figure 4.9). The performance decreases for more than 1024 UPC threads because of the communication pattern. The benchmark reads 25 points of the neighboring 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 [54]. The MPI version is faster but requires additional code before and after calculating the force and moving objects. The compiler does not create additional calls for accessing the data unlike with the UPC versions.

Guppie

The Guppie benchmark uses a 256 MByte array with one billion updates per UPC thread. The random elements are selected for Guppie benchmark from a 64K element array of indexes. The compiler optimizes the main kernel of the benchmark, however the access pattern of shared data is randomly and thus, the performance gain numbers have a variation of +/- 10%. In this benchmark, the runtime is able to coalesce from two up to eight different messages, due to the dynamic nature of the shared accesses. Figure 4.10 shows the performance numbers for the Guppie benchmark in MegaUpdates/s using logarithmic scale. The aggregation gives a speedup from 1.1x up to 7.5x compared to the baseline version. Furthermore, manual code modifications allow the compiler to optimize using the remote update optimization. The remote update optimization uses hardware acceleration. Thus, the achieved performance for the manual optimized code is from 4.64x up to

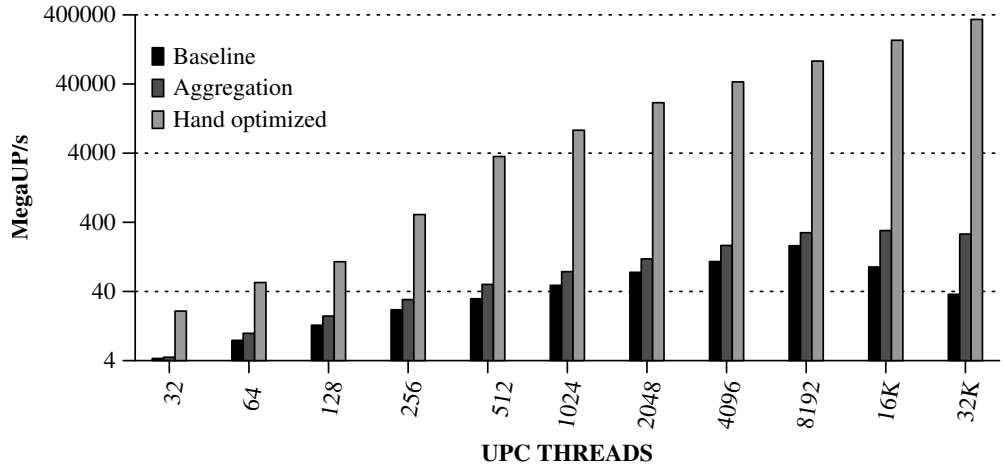

Figure 4.10: Performance numbers for the Guppie benchmark.

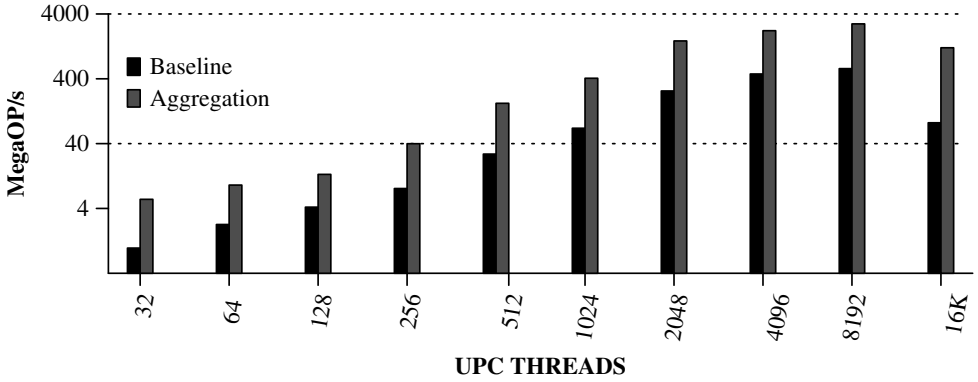

Figure 4.11: Performance numbers for the MCop benchmark.

1270x times faster than the automatically optimized version. The hardware solution provides even more efficient solution than the inspect-executor optimization. Moreover, the hardware packet aggregation give an additional performance gain to the remote update.

MCop

The benchmark solves the matrix chain multiplication and contains irregular shared references. Figure 4.11 presents the performance numbers for the MCop benchmark in Operations/s. The aggregation gives a speedup raging from 4.9X up to 14.5X compared with the baseline version, due to the low performance of the baseline. The automatic compiler optimization is faster due to a better overlap of communication and computation.

4.3.4 Where does the time go?

An interesting question arises: where does the time go during the execution of the application. Figure 4.12 presents the breakdown for the three benchmarks. The shared pointer arithmetic (*Ptr Arithmetic*) translates the offset to the relative offset inside the thread. The inspector loops take 31% and 18% of the execution time in the Sobel and gravitational fish benchmarks, respectively. The trend is similar for the gravitational fish and WaTor benchmarks. The compiler optimizes only the force calculation part in the WaTor benchmark. Thus, the relative contribution of the other parts of the benchmark to the execution time increases. One interesting characteristic of the fish benchmark is the poor performance of the scheduling algorithm (*Schedule*) which is due to the all-to-all communication pattern. Thus, a better scheduling algorithm that would exploit collective communication is needed in order to achieve good performance in the fish benchmark (Figure 4.8).

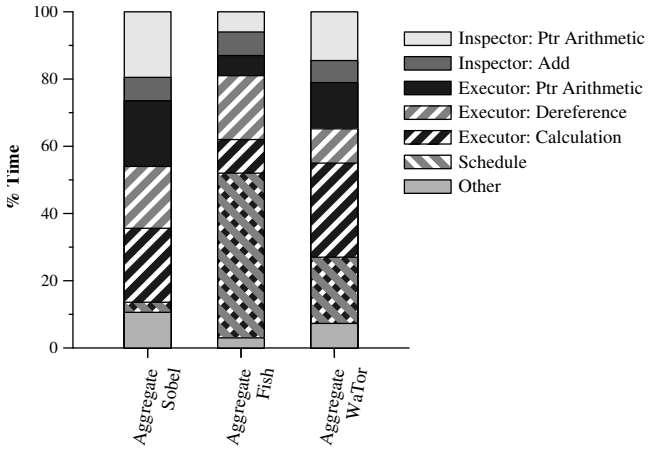

Figure 4.12: Normalized execution time breakdown of the benchmarks using 128 UPC threads.

4.3.5 Cost of the optimization

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

The increase in compilation time varies from 20% to 35%. The main reason for such a large 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 second drawback of the transformation is the increase in code. The code-size increase provides an insight about the glue code that compiler generates. The transformation requires the creation of three additional loops and the strip mining

Benchmark	Baseline	Aggr.	Shared Accesses	Num. Calls	Cost per Access
Sobel	10332	16188	8	38	+732
Fish Grav	22175	28543	4	22	+1592
WaTor	129592	131736	2	14	+1072
Guppie	5022	7134	1	10	+2112
MCop	12188	19100	6	48	+1152

Table 4.2: Object file increase in bytes. We consider only the transformed file.

of the main loop. Moreover, it inserts some runtime calls to the end of inspecting and managing the shared accesses. Table 4.2 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, due to its large number of shared accesses. The number of calls can be calculated using this equation:

$$number\_of\_calls = number\_of\_accesses \times 4 + loops\_optimized \times 6$$

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 optimizes prefetches only one element. On the other hand, MCop has the biggest increase because the compiler optimized four different loops.

Finally, the optimization increases the memory requirements. The runtime keeps information about the shared accesses and uses local buffers for fetching the data. We are aware of the key role of memory usage in scaling to thousands of UPC threads and also acknowledge that the increased memory requirements can limit the performance gain of the application. To address this challenge, we limit the value of the prefetch factor to avoid the allocation of large amounts of memory. For the experiments that we presented, we limit the factor to 2048, although it is configurable. However, large values of the prefetch factor increase the memory footprint of the application and the overhead of analyzing the shared accesses. We limit the additional allocated memory in less than four MBytes because we want the local buffers and metadata to be in the cache hierarchy.

4.4 Chapter Summary and Discussion

This chapter described code transformations that improve the performance of fine-grained communication using data aggregation (or coalescing) at runtime. The results presented indicate that the inspector-executor optimization dynamic communication coalescing is an effective technique decreasing the impact of fine-grained accesses and increasing the network efficiency. This approach can provide better

performance and does not require knowledge of physical data mapping or the usage of the *upc_forall* loop structure.

Despite the high gain in performance that our approach ensures, there is still room for improvements. First, there is the overhead of the inspector loop and the analysis. Although, the transformation amortizes part of the overhead using the double buffering, further improvements can be achieved. One of the overhead sources is the increased number of runtime calls created by the inspector-executor transformation. Second, the absence of collective communication limits the performance in certain data-access patterns. The inability to discover collective access patterns limits the performance in some benchmarks, such as the fish gravitational. Finally, the performance of the random access pattern for more than 2048 UPC threads is limited by the characteristics of the interconnection network i.e. the number of remote links that connect the supernodes. We will address these shortcomings in the following chapters. The relative cost of the optimization, in terms of compile time and code increase, is small compared with the performance benefit.

Chapter 5

Reducing the Runtime Calls

One of the inherited limitations of PGAS languages is the transparency provided to the programmer. Accesses to shared memory lead to the automatic creation of additional run-time calls and possible network communication. The automatic transformation of the code by the compiler solves the problem, delivering the promised ease-of-programming of PGAS language while also delivering the performance expected from a parallel implementation of an application

The previous chapter presented an implementation of shared data aggregation at runtime using the inspector-executor transformation. This chapter presents a number of optimizations to decrease the number of runtime calls in the context of the inspector-executor transformation. First, this chapter presents techniques for decreasing the overhead of inspector and executor loops, include an analysis based on Constant-Stride Linear Memory Descriptors (CSLMADs), usage of a temporary vector to collect the shared indexes, and static coalescing of shared structures. Furthermore, Section 5.1.5 presents the insertion of branches (inline checks) for checking data locality. Section 5.2 presents a new shared-reference-aware loop-invariant code motion, which is designed specifically for PGAS languages. Section 5.3 presents an extensive evaluation of the optimizations using microbenchmarks and benchmarks. Finally, Section 5.4 summarizes previrous discussion and concludes the chapter.

5.1 Inspector-executor Optimizations

An important drawback of the inspector-executor transformation is the overhead of function calls introduced by the compiler in order to inspect which data transfers are amenable for coalescing. This section presents code transformations that help to decrease the overhead of inspector-executor loops.

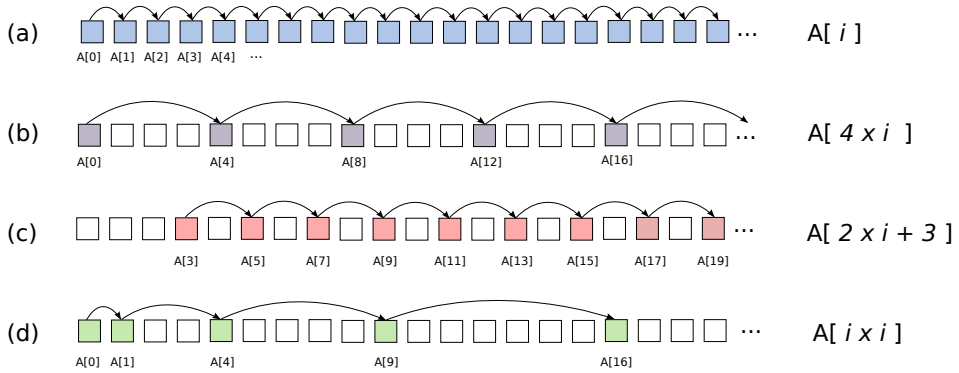

Figure 5.1: Examples of array accesses that Linear Memory Access Descriptors can represent. The CSLMADs can represent (a), (b), and (c) but not (d).

5.1.1 Constant Stride Linear Memory Descriptors

The compiler first analyzes the type of accesses. If the compiler detects that the access on a shared array are regular, it replaces the calls in the inspector loops with a single call. Otherwise, irregular accesses are assumed. The identification of the type of access is done through an array access analysis based on the Constant-Stride Linear Memory Access Descriptors (CSLMADs) [39].

The CSLMADs is restrictive form of Linear Memory Descriptors [40, 41] used on describing the array accesses. The reason we use the CSLMADs rather than the general LMAD description is because the it is more practical to implement the analysis pattern in the compiler. Figure 5.1 illustrates different types of accesses that LMADs can represent. The biggest drawback for the CSLMADs is that they cannot represent arrays array access such as $B[i \times i]$ (Figure 5.1(d)).

CSLMADs is an efficient way to capture accurate array access information. Each array access 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:

$$\langle a, local\_offset, low\_bound + b, upper\_bound + b \rangle$$

The constants (*local_offset*) are used to access each referenced field of the structs.

When the regular access pattern is recognized, multiple calls in the inspector loop can be replaced with a single call that describes the accesses (C.1). The transformation requires a normalized loop with monotonically increasing loop index. Furthermore, if shared arrays were allocated in blocked fashion, then the calls from the executor loops can be removed (C.1.1). In the UPC language, the

compiler detects blocked allocation in two cases: (i) when the programmer specifies the number of threads at compile time, and the blocking factor is the size of the array divided by the number of UPC threads; and (ii) when the programmer uses a structure that contains an array. If the array is allocated with an 'ideal' blocking factor (*e.g.* the size of the array divided by the number of threads), then the fetched data are placed in order. Figure 5.2 presents an example of translating shared addresses to a local buffer. The program fetches the range from 5 to 19. In the top of the figure, the runtime fetches and places the data in order onto the local buffer. Thus, the program reads the data in order. On the other hand, when the blocking factor is two, the runtime fetches the data but the placement is shuffled (bottom of the Figure 5.2).

Compact Runtime Representation

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 merges 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. Thus, the overhead of the inspector loop range is an update of the block ranges. The downside of this approach is that *schedule* calls are necessary in order to wait for the data transfers to complete before the executor loops.

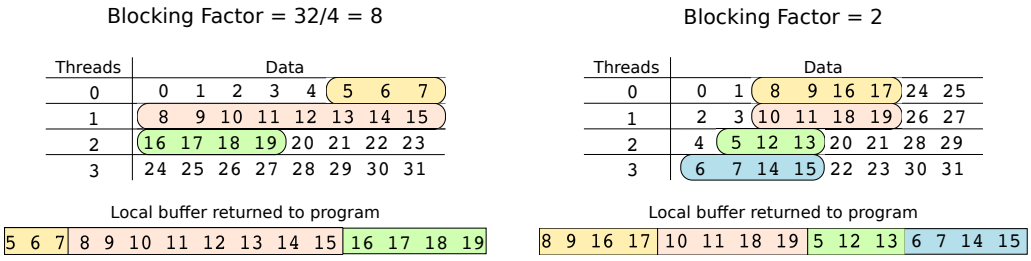

Figure 5.2: The shared address translation problem. The program access the data range from 5 up to 19.

5.1.2 CSLMADs in dynamic environments

Another challenge that the compiler must address is the usage of shared pointers with different blocking factor in dynamic environments, when the number of threads is specified at runtime. In this case, the compiler versions the executor

loops into two variations (C.1.2). The first assumes that the loop has blocked allocation and the second that the loop has blocking factor different than the ideal. A drawback is that the compiler must add checks in the version branch to verify that all the arrays accessed are in blocked fashion. Thus, when any of the arrays has a blocking factor, which is not the ideal, the program executes the loop with the calls.

5.1.3 Usage of vectors

The compiler analyzes shared accesses that occur in an irregular fashion to check if they access more complex structures, such as shared arrays contains structs. For irregular accesses on shared arrays that do not contain structs, a temporary array (vector) stored in the stack is used to collect the shared indexes (C.2). The compiler inserts a call to inspect the elements at the end of the each inspector loop. Internally the runtime processes the elements one by one. Thus, the performance gain is limited compared to the previous approach. The main benefit of this solution is the decrease of the number of calls in the inspector loop. The downside of this approach is that it allocates space in the stack and, when the available stack memory is low, the application can crash.

5.1.4 Combining Dynamic with Static Coalescing

The final check of the compiler involves the possibility that the loop contains shared accesses on struct fields that have constant distance or stride. The algorithm coalesces shared accesses when the compiler can prove that the remote data belongs to the same thread. This is also possible when accessing members of shared structures that belong to the same thread. Therefore, the compiler applies this optimization (C.3.1) when the program uses shared arrays with data structures. The compiler analyzes the accesses and detects possible patterns with constant-stride.

The static coalescing requires additional compiler analysis and runtime modifications. Algorithm 3 presents the analysis algorithm. First, the compiler analyzes the shared accesses that are fields of shared structures. The compiler classifies the shared addresses in buckets containing compatible shared addresses (line 6). A shared reference is compatible with one bucket when the containing shared references use the same base symbol (array), the same array index, same element access size, but different offset inside the structure. If the compiler is not able to find any compatible bucket, then it creates a new bucket and adds the shared reference (line 14). Finally, the compiler sorts the shared references during the addition to the bucket, based on the local offset.

Algorithm 4 presents the insertion of *__xlupc_sched_dereference* 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.

```
AnalyseSharedRefs(Procedure p)
 1: RefList ← collectSharedReferences();
 2: BucketRefList ← ∅;
 3: for each shared mem ref R_s in RefList do
 4:    isInserted ← 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 ← TRUE;
 9:          break;
10:       end if
11:    end for
12:    if isInserted = FALSE then
13:       Bck_s ← newShrReferenceBucket();
14:       Bck_s.Add(R_s);
15:       BucketRefList.Add(Bck_s)
16:    end if
17: end for
```

Algorithm 3: Analysis of shared references.

```
InsertSchedulerDereferenceCalls(Procedure p)
 1: for each Shared Bucket bck_s in BucketRefList do
 2:    R_base ← bck_s.getBaseSharedReference();
 3:    stmt ← innerloop_i.findLocation(R_base);
 4:    innerloop_i^stmt.Add( buffer = __xlupc_sched_dereference, &index);
 5:    for each shared mem ref R_s in bck_s do
 6:       stmt_s ← 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 ← index + 1;
13:    end for
14:    stmt ← innerloop_i.findLocation(R_base);
15:    innerloop_i^stmt.Add( buffer = __xlupc_sched_dereference );
16:    for each shared mem ref R_s in bck_s do
17:       stmt_s ← 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 ← index + 1;
24:    end for
25: end for
```

Algorithm 4: Insertion of dereference calls.

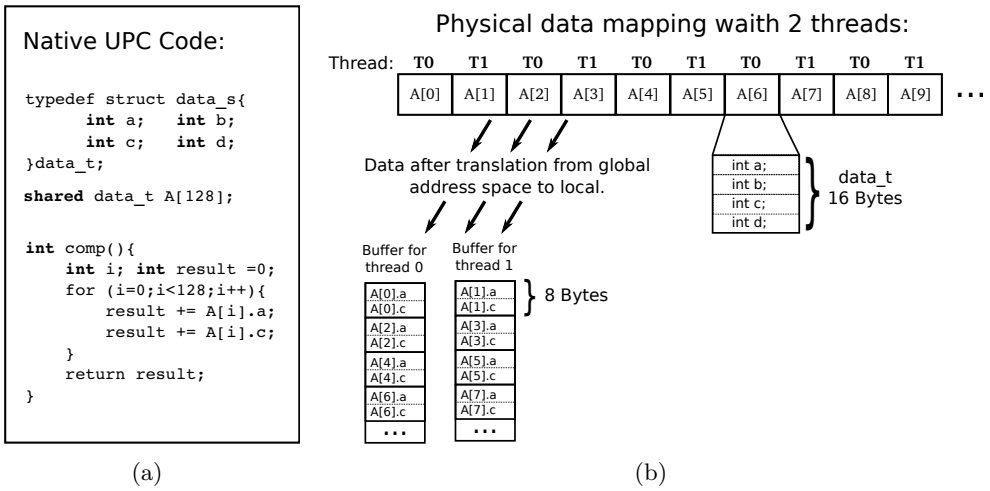

Figure 5.3: Example of Static data coalescing: native UPC source code (left), and physical data mapping (right).

Figure 5.3 exemplifies the static coalescing optimization. The example program on the Figure 5.3(a) shows a simple reduction of the `a` and `c` struct fields, from a shared array of structures written in UPC. Figure 5.3(b) presents the physical mapping of the shared array running with two UPC threads. The array is distributed cyclically among the UPC threads. The transformation places the accesses on the `a` and `c` struct fields in the same bucket. 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 in the bucket. At runtime, the runtime fetches the `a` and `c` fields and places them in consecutive memory locations in the local buffer. On the other hand, if two consecutive are accesses, then the compiler would still represent it a stride with size of the element.

Local data optimization

One of the key optimizations of the runtime is the efficient access management of shared data that belong to the same node. The runtime collects and analyzes the remote shared references only. Furthermore, the runtime avoids copying the data in local buffers in order to provide the same memory layout. In the dereference call, the runtime simply returns a pointer to the local data.

However, this approach requires symmetrical physical data mapping between buffers. To address this challenge, the dereference call returns to the program the constant stride between the offsets. Figure 5.4 presents the generated code and a part of the dereference call in the runtime. The compiler first sets the default stride, if the data are local, before the call with the distance between accesses. In the aforementioned example, the distance between the fields is eight bytes. The

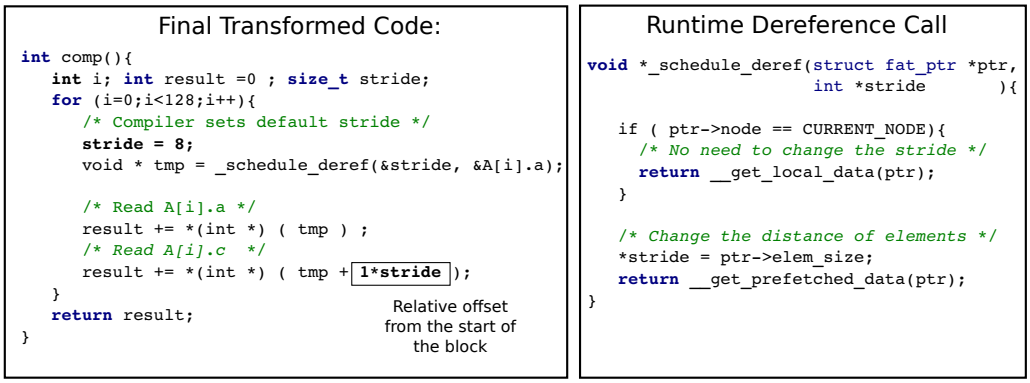

Figure 5.4: Final code modification and a high level implementation of the runtime.

runtime sets stride variable as the element stride when the data are prefetched and stored in the local buffer. On the other hand, if the program access local data, then the stride remains constant inside the runtime and the default is used. The compiler multiplies the relative offset based on the order of shared references.

5.1.5 Inline checks

```
1    shared int A[128];
2
3    // Inspector loop. shared ptr is fat pointer
4    for (i=0;i<PF;i++){
5        ptr = __ptr_arithmetic(&A[i]);
6        if ( ptr.thread != MYTHREAD ){
7            __xlupc_add_access( ptr, ...);
8        }
9     }
10
11     __schedule();
12
13   // Similar approach in the executor loop:
14   for (i=0;i<PF;i++){
15        ptr = __ptr_arithmetic( &A[i],... ):
16        if ( ptr.thread != MYTHREAD ){
17           local_ptr = __xlupc_sched_dereference( ptr, ...);
18        } else {
19           // Simple pointer additions
20           local_ptr = CALC_LOCAL(ptr);
21        }
22        ... = local_ptr;
23     }
```

Listing 5.1: Example of code modifications for the inline checks.

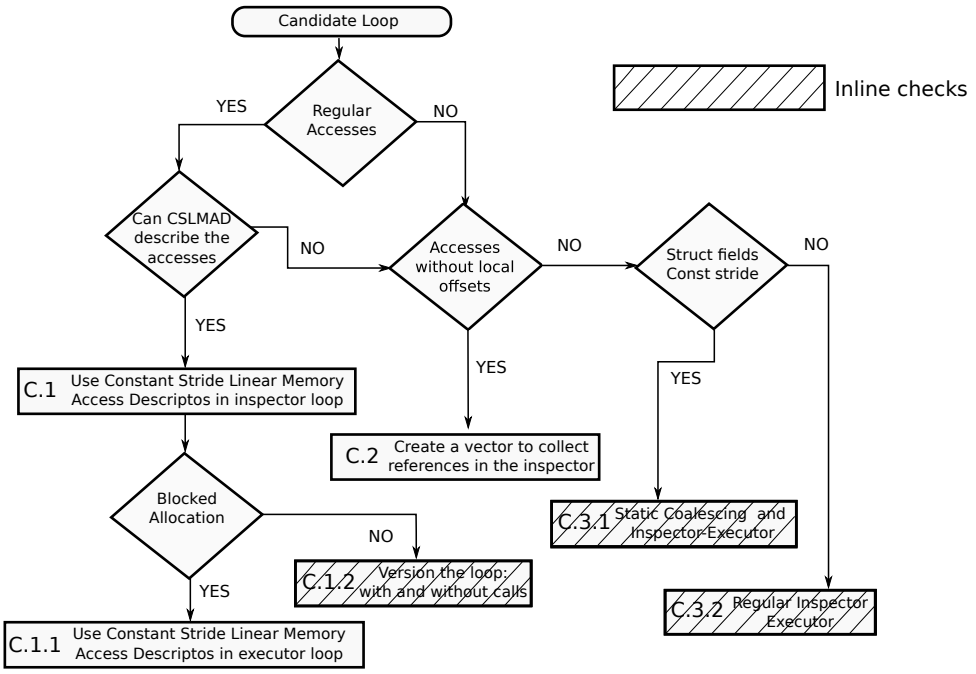

Figure 5.5: Improvements for the for the inspector-executor compiler transformation.

A number of UPC applications have local shared references that cannot be proven to be local shared references at compile time. In applications that have good locality only a small proportion of shared accesses are remote. For instance, in the Sobel benchmark [50] only 1.6% of the shared accesses are remote when running with 2048 UPC threads.

The inline check optimization inserts a branch before two entry points of the runtime: the *__sched_add_access* and the *__xlupc_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 private pointers to read the local data. Listing 5.1 presents an example of the code transformation.

5.1.6 Optimization Integration

Figure 5.5 presents the algorithm that compiler uses to optimize loops with fine-grained accesses. The algorithm identifies the pattern of the accesses pattern (regular or irregular) and uses compile time information to select the proper transformation. If the accesses are regular, the compiler analyzes the accesses to detect the format of access pattern (C.1) and removes the runtime calls from the inspector loops. Next, based on the compile time information it removes completely (C.1.1)

the runtime calls from the executor or versions the loop (C.1.2). In the case of irregular accesses, the compiler creates a temporary array to collect the elements (C.2). Finally, if the programmer uses structures with fields, the compiler tries to apply static coalescing (C.3.1) or applies the simple form of inspector-executor (C.3.2). Furthermore, the compiler inserts inline checks in some for the cases to check for data locality. Next sections explore these cases in more detail.

5.2 Shared-reference-aware loop-invariant code motion and privatization for PGAS languages

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. As a result, some shared accesses are left inside the body of the loop. Furthermore, copy propagation interferes with loop-invariant code motion because the existence of shared variables is often ignored. For instance, if a statement that uses a shared variable is propagated and placed inside a loop, later the compiler replaces the shared accesses with runtime calls.

```
1  #define SIZE 1024
2  struct OceanGrid{
3    int FishSmell;
4    int SharkSmell;
5    ...
6  };
7
8  shared [*] struct OceanGrid Ocean[SIZE][SIZE];
9  shared int MaxfishSmell;
10 shared int MaxsharkSmell;
11 ...
12 for(di=0; di<5; di++){
13   for(dj=0; dj<5; dj++){
14     int idx = (SIZE+i+di-2)%SIZE; int idy = (SIZE+j+dj-2)%SIZE;
15     if( Ocean[idx][idy].FishSmell > MaxfishSmell) {
16         MaxfishSmellPosX = di-2;
17         MaxfishSmellPosY = dj-2;
18       }
19     if( Ocean[idx][idy].SharkSmell > MaxsharkSmell) {
20         MaxsharkSmellPosX = di-2;
21         MaxsharkSmellPosY = dj-2;
22       }
23   }
24 }
25 ...
```

Listing 5.2: One of the Wator ocean simulation kernels.

Listing 5.2 presents one of the kernels of the Wator [73] benchmark. The kernel calculates the maximum smell of the fishes and sharks. The two branches (lines 15, 20) contain two different shared accesses each. The first access is on the shared array of the Ocean and the indexes depend on the induction variables of the loops. On the other hand, the scalar variables `MaxfishSmell` and `MaxsharkSmell` are shared. The compiler assumes they are simple scalars, and it does not move them outside of the loop, despite that they are shared.

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.

Algorithm 5 presents this new loop-invariant code motion. First the algorithm checks if the loop qualifies for this transformation. The loop must be normalized and must contain shared references. For each shared reference, the algorithm uses the reaching-definition analysis through the Static Single Assignment (SSA) [42] representation. In addition, the algorithm also checks the upper bound of the loop for possible shared references. Finally, the algorithm checks for possible dependencies between loop iterations. For each independent shared reference, the algorithm stores the shared value to a temporary scalar variable before the loop. Then, the algorithm replaces all the occurrences of the shared references inside the body of the loop.

5.3 Experimental Results

This experimental evaluation assesses the effectiveness of the modified code transformations through the following: (1) the performance on microbenchmarks to help understand the maximum speedup that can be achieved and the potential performance bottlenecks; (2) the comparison with the sequential C version; (3) the performance of real applications; (4) the impact of the number of iterations examined; (5) an analysis of the overhead observed; (6) measurements of the code-transformation cost in code increase, compilation time, and code length.

5.3.1 Methodology

All runs use one process per UPC thread and schedule one UPC thread per Power7 core. There are UPC threads in 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 3% 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 is running during the measurement. All benchmarks are compiled using the '`-qarch=pwr7 -qtune=pwr7 -O3 -qprefetch`' compiler flags in order to enable Power7-specific

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

Algorithm 5: UPC loop-invariant code movement

code transformations. The evaluation tries to keep the computation constant per UPC threads (weak scaling). In Sobel, Fish, and WaTor benchmarks, the evaluation doubles the dataset every quadruplication of UPC threads. 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 disabled.

Aggregation: compiled with the inspector-executor code-transformation that prefetches and coalesces shared references at runtime.

Aggregation Optimized: combines the inspector-executor transformation with the improvements presented in this chapter and 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.

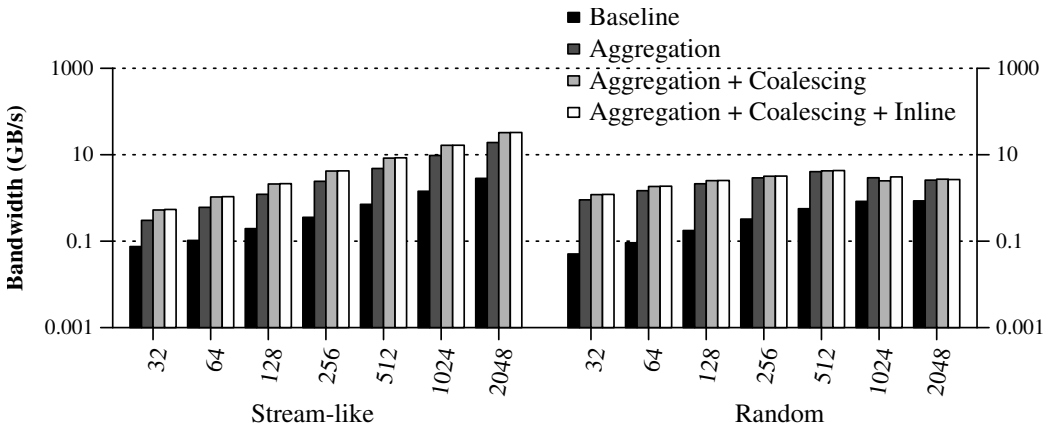

Figure 5.6: Performance in GB/s for the microbenchmark reading four fields from the same data structure reading four fields.

MPI: contains coarse-grained communication and uses collective communication whenever possible. This version uses blocking calls for communication but it does not use the one-side communication model introduced in MPI 2.0.

5.3.2 Microbenchmark Performance

Microbenchmanks are created to demonstrate the effectiveness of the code transformations. Indeed, the results presented in Figure 5.6 — notice the log scale in the graphs — confirm that, when applied to the code that they target, the code transformations are very effective. In `stream-like` microbenchmarks, 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 transformation inserts a branch before two entry points of the runtime: *__sched_add_access* and *__sched_dereference*. These branches check if the data accessed are remote or local, and minimize the overhead of accessing local data as described in section 5.1.5.

`Stream-like` contains only remote shared references because it accesses the next neighbor thread. Thus, the inline branch to avoid calls when the shared data are local is always taken. In contrast, `random-access` results in a speedup between 3.2x and 21.6x from prefetch with an additional 4-8% due to struct-field coalescing. In contrast with `stream-like`, the inline transformation gives an additional 2-5% performance gain because it is able to improve local shared accesses.

An interesting observation emerging from the results is that `random-access` achieves better bandwidth than `stream-like` when the code transformations are

App	Dataset	Sequential C	UPC Single-Thread		32 UPC Threads	256 UPC Threads
			Static	Dynamic		
Sobel	64K x 64K Image	101.0	93.5	110.1	3.6	0.7
Fish	16K Objects	155.7	642.8	727.7	24.8	7.2
WaTor	4K x 4K Grid	9.8	48.3	791.3	98.2	28.1
Guppie	2^{28} Elements	70.5	305.1	349.6	104.3	22.1
MCop	1024 Arrays	16.6	19.2	29.3	140.8	22.0

Table 5.1: Benchmarks compared with the serial C non-instrumented version and UPC version in execution time, measured in seconds.

enabled. This results from the random traffic pattern in combination with the high-radix interconnect when using direct routes. Previous research on the PERCS interconnect architecture confirms the experimental results [81].

5.3.3 UPC Single-Threaded Slowdown

This section, prior to the scalability measurements, studies the performance of UPC language compared with the serial version. Thus, this section examines the single-thread overhead of an UPC program. The UPC language offers a programming model for distributed systems and programs are designed to run in thousands of cores. The single-thread overhead, shown in Table 5.1 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. For the serial version we use the `gcc` version 4.3 that was available on the machine. The most important reason that increases the single-thread overhead is the need to use *fat pointers* to access data into distributed arrays. The runtime call for the address translation in a fat pointer is needed because, without knowing the number of threads, the compiler cannot determine whether the memory accesses are to the single local thread. The runtime in the single-threaded dynamic column in Table 5.1 is obtained by running binaries where the number of threads is not known at compile time. The numbers under the static column are the execution times when the compiler has the information that the program will execute on a single thread, and thus can eliminate the runtime calls for address translation. For instance, the single-thread slowdown for `WaTor` is reduced from $81x$ to $5x$. The runs with 32 and 256 UPC threads are 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 due to the compiler limited ability 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. Firstly, it has good

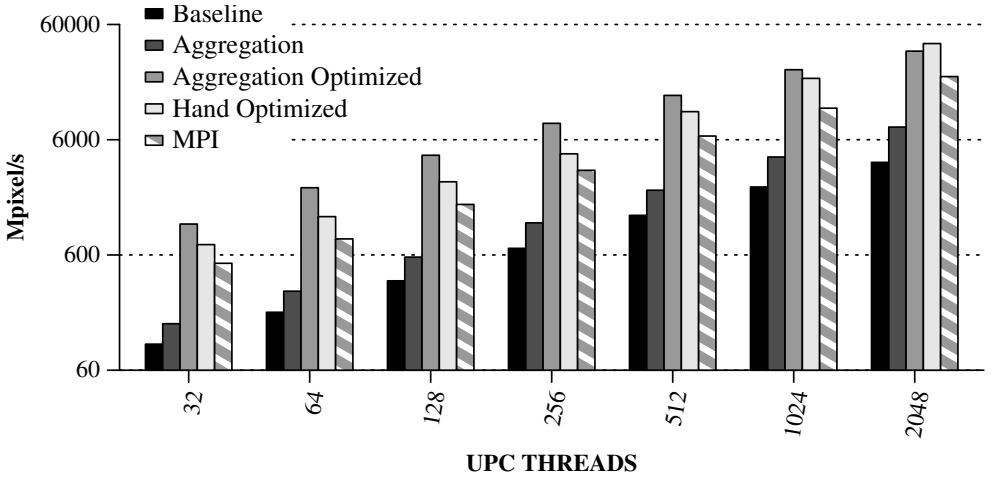

Figure 5.7: Performance numbers for the Sobel benchmark for different versions.

shared data locality; it fetches data only from the neighboring threads. Secondly, the compiler removes the calls completely from the inspector and executor loops. The low performance in the single-thread version is because the program executes the unoptimized version of the loop to avoid the overhead of the shared accesses analysis. Furthermore, the performance of the serial Sobel version is slightly slower than the UPC version because we are using two different compilers with different codebase.

These results indicate that one order of magnitude slowdown is typical when converting a program from C to the UPC language. This slowdown underscores the importance of the code transformations that remove unnecessary runtime calls automatically. The focus of this chapter is to combine the removal of these calls with more traditional code transformations, such as the inspector-executor transformation and the loop-invariant code motion that was adapted to PGAS languages. A point to take is that PGAS programmers and compilers should not focus only on reducing the cost of communication, but also in reducing the overhead of runtime calls.

5.3.4 Applications Performance

This section explores the performance of the code transformations when applied to benchmarks. As described in Table 3.3, runtime calls can be removed only in **Sobel** and **Fish** because those are the only benchmarks that contain regular accesses. The compiler successfully removes some of the calls in **MCop**, but not all of them. **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 compiler also applies the inline code transformation on all these benchmarks.

Sobel achieves a performance gain between 1.5X and 2X using the inspector-executor (*prefetch*) code transformation as shown in Figure 5.7 The *prefetch opti-*

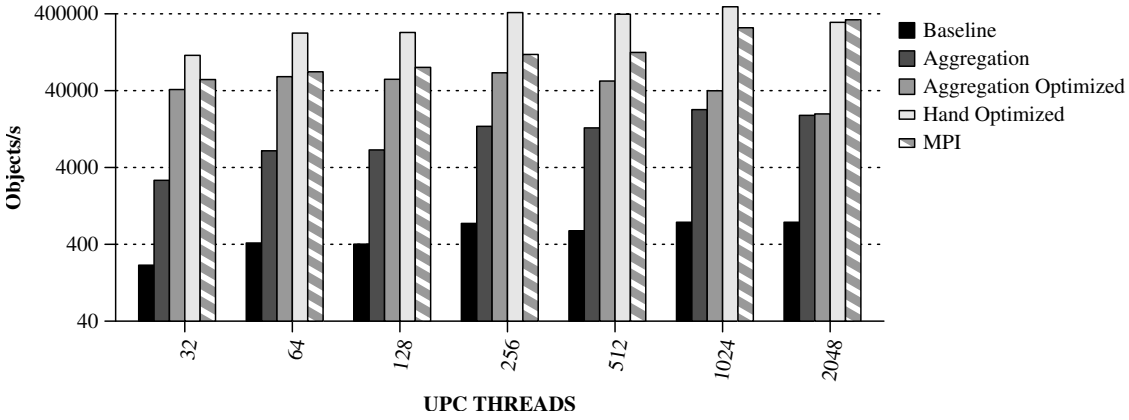

Figure 5.8: Performance numbers for the fish benchmark for different versions.

mized technique achieves from 9.2X up to 12.3X speedup over the *baseline* because the CSLMAD representation completely removes the calls. The *hand optimized* UPC version is faster than the *MPI* version due to the one-side communication. However, the performance of the *hand optimized* and the *MPI* versions are converging with more than 1024 UPC threads. One interesting observation is that the *prefetched optimized* version is faster than the UPC hand-optimized one due to double buffering. Unfortunately, the current version of UPC language does not support asynchronous memget/memput calls. Thus, the advantage of compiler transformations is the automatic overlapping communication and computation.

The **fish benchmark** exhibits high performance gains because the *baseline* is inefficient as shown in Figure 5.8. The compiler uses the CSLMADs to remove the runtime calls from the inspector and executor loops. However, 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 inspector-executor and loop-invariant code motion transformations.

In comparison, the performance for the **WaTor benchmark** is lower: the *prefetch optimized* version is 1.12X to 1.72X faster than the *baseline* (Figure 5.9). The explanation for this result can be found in the transformation of a loop structure that has a constant number of iterations (25): the force calculation computation. The force is calculated using a 25-point stencil, looking at the 5×5 neighborhood of each grid point. The benchmark uses the fish and shark smell to calculate the force. The pray (fish) get a negative force from the grid point (cell) that contain predator (shark) smell or a random force if no sharks are present. Sharks get a positive force towards to the fish smell or towards to the center of mass of the fish if no fish smell is present.

The compiler improves the performance of the remaining fine-grained shared

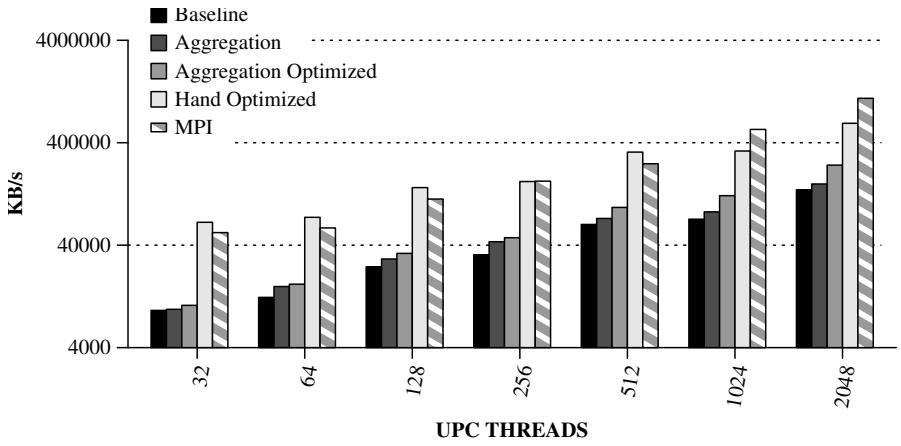

Figure 5.9: Performance numbers for the WaTor benchmark for different versions.

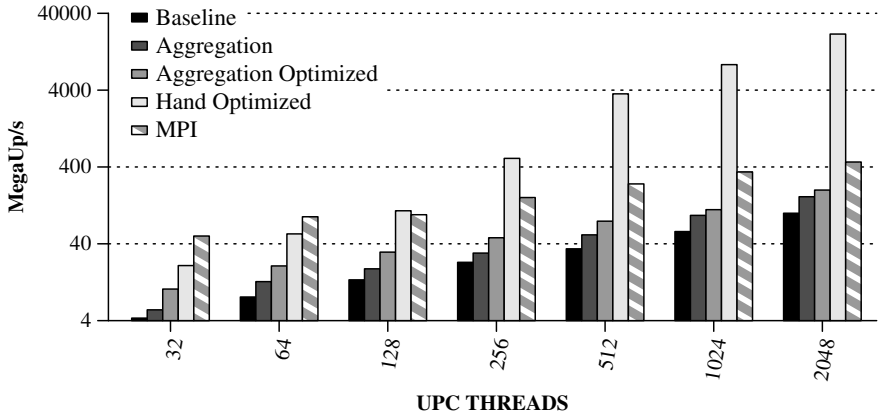

Figure 5.10: Performance numbers for the Guppie benchmark for different versions.

accesses by using the remote update runtime calls elimination [54]. The *MPI* version is faster but requires additional code before and after the calculation of force and movement of objects. The *MPI* programming model requires the privatization of the accesses. Moreover, the programmer must explicitly express the data transfers between different MPI processes or nodes. In the case of the WaTor benchmark, the programmer inserts MPI communication primitives before and after each phase of the simulation. The downside of this approach is that the programmer must design carefully the data transfers taking account the physical data mapping.

The **Guppie benchmark** uses 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 1.6X and 2.53X speedup over the *baseline* (Figure 5.10). The compiler manages to remove the calls from the inspector loops thus decreases the overhead of function calls and

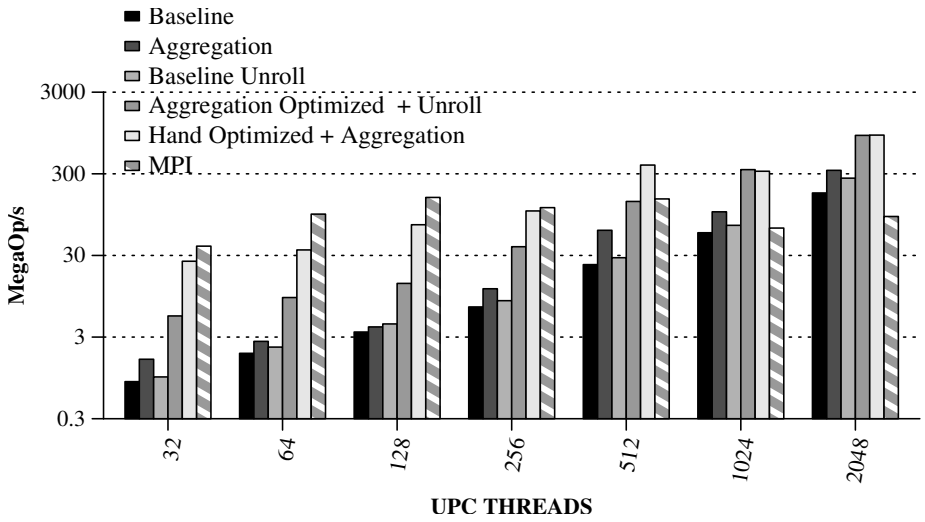

Figure 5.11: Performance numbers for MCop benchmark for different versions.

shared-pointer arithmetic. It is known that manual code modifications to this benchmark can allow a compiler to improve the performance be eliminating remote updates [54]. The benchmark uses a temporary buffer to fetch the data, modify them, 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 and uses hardware acceleration including packet aggregation to perform the updates. Thus, the achieved performance for the manually modified code is one order of magnitude faster than the automatically optimized version. The difference increases with the number of UPC threads.

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 UPC versions for a small number of threads. On the other hand, the manual UPC optimized version is 46X times faster than the MPI version running with 2048 Threads. This result provides strong evidence in support of the key role of the remote update code transformation. 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 1.6X up to 2.6X compared with the baseline version in the **MCop benchmark** as shown in Figure 5.11. Applying the code transformations and manual unroll the loops four times gives a speedup from 4.9X up to .3X. Despite the removal of most calls in the loops, there are still irregular references. The *manual 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 in magnitude faster than the MPI version.

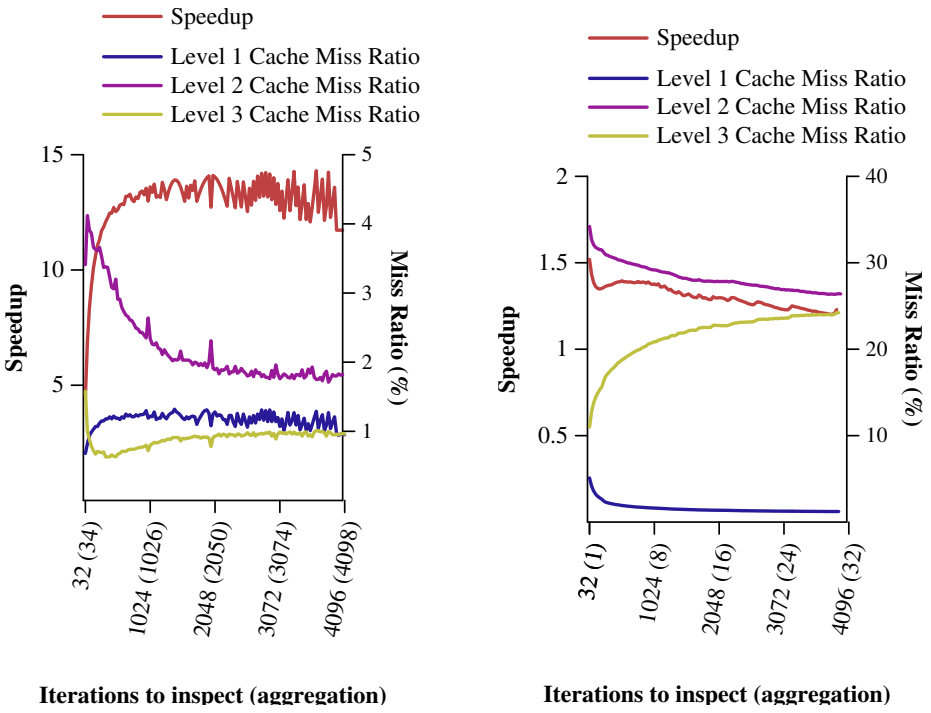

Figure 5.12: Speedup and cache misses for `Sobel` (a) and `Guppie` (b) using different number of iterations to inspect and aggregation levels.

5.3.5 Parameter exploration

An important question is how many iterations the runtime should prefetch. Previous work using inspector-executor code transformations [38, 82] suggest that this number should not be too large. The main problem with prefetching too many iterations is the interference with the cache memory of the processors. Figures 5.12(a) and 5.12(b) present the speedup and the local cache miss ratio [1] 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, each UPC thread makes 16777216 updates. In total 99.21% of the shared accesses are remote. The figures also show the average number of aggregated messages in the parenthesis of x-axis. The number of aggregated messages in `Sobel` is the number of iterations to inspect plus two for the border of the image. On the other hand, the number of the messages aggregated in `Guppie` is calculated using the equation: $\#Iterations\_to\_inspect/UPC\_THREADS$, because of

[1] Local miss ratio is calculated by diving misses in this cache the total number of memory accesses to this level of cache.

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 correlates 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, the limit of the application is not the network communication as explained in section 5.3.6.

In contrast, `Guppie` incurs high miss ratio due to the random access pattern that it employs. In this case, despite the larger number of aggregated messages, the speedup decreases for two reasons: (1) because there is an increase in the L3 miss ratio due to the allocated buffers; 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.

5.3.6 Overhead Analysis

This section examines the remaining overheads of UPC applications and responds to the previously asked question of 'where does the time go' during the execution of the application. Two representative benchmarks are selected for this evaluation. Namely, the `Sobel`, which contains regular access and `Guppie`, which contains random accesses. Figure 5.13 presents a breakdown of the normalized execution time before and after the code transformations. The runs use the Linux Perf Tool [83] to collect performance counters.

Using the inspector-executor approach in `Sobel`, the time devoted to the computation decreases significantly. One interesting characteristic of `Sobel` is that it spends more than 55% of the time in the shared pointer arithmetic in the Prefetch version, due to additional calls in the inspector loops. The shared pointer arithmetic translates the shared index to the virtual address 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 `Sobel`, but is still important. 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 reduces the communication overhead down to 57%. However, the overhead is transferred to the shared references analysis, due to the irregular communication pattern. Therefore, the improved transformation successfully hides the overhead from the programmer's code, but the runtime still processes the elements one by one. The inline code transformation has a minor impact on the achieved performance, and it is only visible for a lower number of threads.

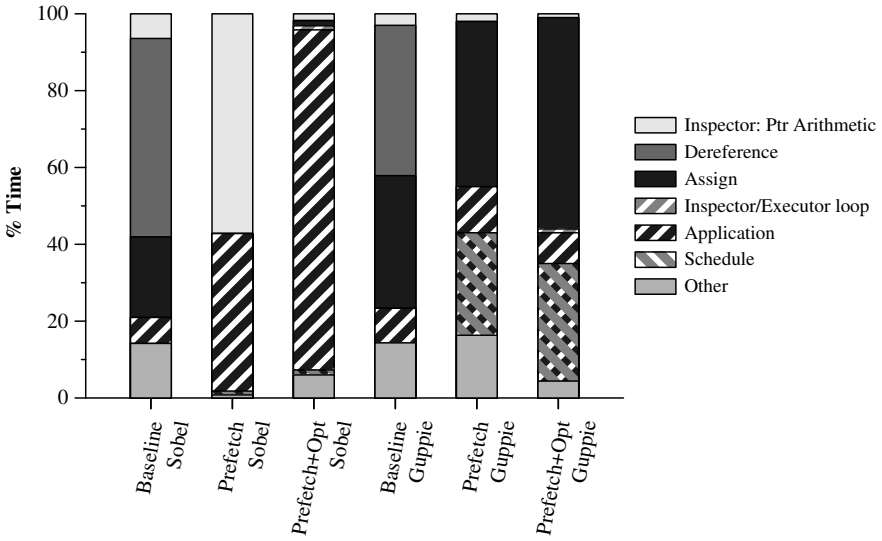

Figure 5.13: Normalized execution time breakdown of the benchmarks using 32 UPC threads.

5.3.7 Compilation Time and Code Length

This section examines the transformation cost in compile time, code-size increase, and productivity in lines of code. Although, measuring lines of code is not always the best metric of productivity cost [84], it provides a good indication about the developing cost. The code-size increase provides an insight about the glue code that the compiler generates. The compilation time is another important metric that is of concern to the programmer. Table 5.2 presents the increase of the code, the compilation time, and the number of lines of code for the five benchmarks.

The increase in compilation time varies and depends on the code transformations applied and on the number of potentially remote accesses in the benchmark. The main reason for such a large increase is the rebuild of data and control flow graph of the transformed loops. The number of shared accesses affects the compilation time. The programmer can tolerate larger compilation times because the cost is amortized with better runtime performance. However, three or four times increase in the compilation time compared with the MPI version reveals the immature state of the UPC compiler.

The second drawback of the inspector-executor transformation is the code increase. The transformation requires the creation of additional loops and the strip mining of the main loop. Moreover, it inserts runtime calls to inspect and manage shared accesses. The Sobel benchmark with the modified inspector-executor code transformation (*prefetch*) approach has the biggest code increase, due to its large number of shared accesses. Although the evaluation uses the Power7 processor, other architectures (*e.g.* embedded systems and GPUs) can also potentially ben-

Application	Version	Size	Compile Time (s)	Source Lines Of Code
`Sobel`	UPC Baseline	2240	10.818	160
	UPC Prefetch	3977	16.066	160
	UPC Hand Opt.	2596	12.298	220
	MPI	1319	6.553	210
`Fish`	UPC Baseline	22879	6.049	246
	UPC Prefetch	15979	6.948	246
	UPC Hand Opt.	22439	5.742	252
	MPI	12253	1.328	251
`WaTor`	UPC Baseline	42468	16.648	792
	UPC Prefetch	44644	20.619	792
	UPC Hand Opt.	43076	19.007	980
	MPI	13447	2.901	761
`Guppie`	UPC Baseline	6274	1.816	176
	UPC Prefetch	8210	2.468	176
	UPC Hand Opt.	5218	1.702	176
	MPI	4572	0.905	610
`MCop`	UPC Baseline	12524	3.011	171
	UPC Prefetch	19324	5.145	171
	UPC Hand Opt.	25820	8.587	204
	MPI	16210	0.649	176

Table 5.2: Benchmarks and different cost metrics. Object file sizes are in bytes and for the object files only.

efit from these code transformations. However, in such architectures the benefits may be limited by memory constraints.

In terms of program length, the baseline versions are the smallest. The overall trend is that the MPI version has the smallest compilation time and the smallest binary output. On the other hand, the MPI programs are longer than the UPC versions. Moreover, the hand-optimized UPC and MPI versions have similar length. Overall, the UPC language provides acceptable performance using the code transformations described in this paper, with reasonable programming effort. On the other hand, achieving the hand-transformed performance for an UPC program containing fine-grained communication requires effort similar to MPI.

5.4 Chapter Summary and Discussion

This chapter described code transformations that aggressively remove the automatically generated runtime calls in order to improve the performance of fine-grained communication. The current prototype implementation can be improved in several ways.

This benchmark-based performance study allow us to predict that through the new code transformations presented applications with regular accesses can achieve between 60% and 180% of the hand-optimized UPC version. This improvement is due to better overlap of computation and communication and advances previous attempts to decrease the overheads of dealing with remote accesses. The evaluation

results support the argument that code transformations should focus on removing the calls completely rather than focusing simply on improving communication or on applying limited privatization, coalescing, and shared-data caching.

On the other hand, there is still room for improvement for benchmarks that contain irregular communication. In this case, the programmer can benefit by expressing the accesses in the form of read-modify-write of data. The compiler can then exploit the benefits of the remote update code transformation. Thus, the effectiveness of the code transformation is high in hardware architectures that support hardware acceleration of remote updates such as the Power 775 [63].

Applications with irregular accesses, which cannot be transformed into the form of a read-modify-write, achieve better performance, but there is still room to close the gap with the hand-transformed and MPI versions. In this case, the programmer should manually privatize the globally shared array in order to achieve good performance.

Chapter 6

Loop Scheduling

The last part of the thesis presents optimizations for improving the network communication and avoid the creation of hotspots when using the UPC language. A typical communication pattern in UPC programs is the concurrent access of shared data allocated to each UPC thread by all the other threads. In addition, *for* loops that contain all-to-all or reduction communication can overwhelm the nodes and create network congestion. Listing 6.1 presents an example of a naive reduction code executed by all the UPC threads. In this example, all the UPC threads execute this part of the code, creating network hotspots. The array is allocated in blocked form, thus the first N/THREADS elements belong to the first UPC thread. The groups of optimizations presented in this chapter focus on code transformations that decrease this effect.

The creation of hotspots has an even higher in high-radix networks such as the PERCS interconnect [66]. High-radix network topologies are becoming common approach [63, 85, 86, 87] to addressing the latency wall of the modern supercomputers, such as the Power 775. The high-radix networks provide low latency though low hop count.

```
1  #define N 16384
2  shared [N/THREADS] int A[N];
3
4  long long int calc(){
5    int sum = 0, i;
6    for (i=0; i<N; i++) // Executed from all UPC threads
7        sum += A[i];
8    return sum;
9  }
```

Listing 6.1: Example of reduction in UPC.

Section 6.1 examines possible solutions to address the problem of hotspot creation. Specifically, we exemplify four approaches to address these problems. Namely, (i) skewing loops in order to start from a different point of loop iteration space; (ii) using skew loops in order to start from a different point of loop iteration

and inside the block; (iii) accesseses the elements with the constant stride; (iv) randomly access the elements. Next, Section 6.1.2 proposes a number of compiler transformations to resolve automatically the problems without the the programmer's interference. Section 6.2 presents the experimental evaluation using (i) a limit study with microbenchmarks to measure the effectiveness of different approaches and (ii) automatic compiler transformation compared with the manually transformed loops.

6.1 Loop scheduling

A difficult communication pattern of UPC programs is the concurrent access of shared data allocated to one UPC thread. In addition, the all-to-all communication is another pattern that stresses the interconnection network. In this case, each thread communicates with all other UPC thread to exchange data. This communication pattern is one of the most important metrics for the evaluation of the bandwidth of the system. Moreover, this pattern shows up in a large number of scientific applications including FFT, Sort, and Graph 500. Thus, it improves the network's efficiency by scheduling the loop iterations, and can significantly decrease the communication overhead. This section examines four different approaches to schedule loop iterations for either coarse-grained or fine-grained communication. The transformations assumes that the programmer allocates the shared arrays in blocked fashion. Furthermore, we assume that the number the loop upper bound has the same value as the number of elements of the loop. This assumption is not always true but it simplifies the presentation of the algorithms. Moreover, it presents a solution to automatic loop transformation.

6.1.1 Approaches

The core idea is to schedule the accesses in such a way that each thread does not access the same shared data. The programmer manually transforms the loop to increase the performance. The loop scheduling transformations are grouped in four categories as follows:

- *Skew loops* to start the iteration form a different point of the loop iteration space. In UPC language, this is made possible by using the `MYTHREAD` keyword in the equation that calculates the induction variable. To calculate the new induction variable of the loop we use the following equation:

$$NEW\_IV = (IV + MYTHREAD \times Block) \,\%\, UB;$$

  Where $Block = \frac{SIZE\_OF\_ARRAY}{THREADS}$ and UB is the upper bound of the loop.

- *Skew loops plus*: The 'plus' is used to uniformly distribute the communication among nodes. Each UPC thread starts from a different block of the shared array and also from a different position inside the block. The new induction

variable is calculated as follows: $NEW\_IV = (IV + MYTHREAD \times Block + MYTHREAD \times \frac{Block}{THREADS}) \% UB$;

Figure 6.1 right shows differences between the two skewed versions. The 'plus' version access elements from other UPC threads in diagonal form. This approach is expected to achieve better performance than the baseline because it uniformly spreads the communication more than the 'simple' version.

- *Strided accesses*: Each thread starts from a different block. The loop increases the induction variable by a constant number: the number of UPC threads per node plus one. This approach requires the upper bound of the loop to not be divisible by the constant number (*stride*) [88, 89, 90]. The new induction variable is calculated by the following equation:

$$NEW\_IV = (IV \times STRIDE + MYTHREAD)\%UB;$$

To ensure the non-divisibility of the loop upper bound we can use a simple algorithm:

$$STRIDE = THREADS + 1;$$

$$while\ (UB\%STRIDE == 0)\ STRIDE++;$$

For example in the Power 775 architecture, when running with 32 UPC threads per node and assuming that the upper bound of the loop is 2048, the new induction variable is calculated by:

$$NEW\_IV = (IV \times 33 + MYTHREAD) \% UB;$$

- *Random shuffled*: the loop uses a look-up array that contains the number of threads randomly shuffled. This approach works only when the upper bound of the loops is equivalent to the number of threads. This loop creates an all-to-all communication pattern through the network. This optimization is applicable to loops with both fine-grained and coarse-grained communication. There are two downsides when applying this approach to loop with fine-grained communication. First it requires $SIZE\_OF\_ARRAY \times NUM\_THREADS \times sizeof(uint64\_t)$ memory that cannot be allocated of large arrays. The second drawback is that the runtime (or the compiler) has to shuffle the array, wasting valuable time of the program execution.

6.1.2 Compiler-assisted loop transformation

The idea of compiler-assisted loop transformation is to conceal the complexity and make the network more straightforward for the programmer. First, the compiler collects normalized loops that contain shared references and have no loop carried dependencies . Next, the compiler checks when the upper bound of the loop is greater or equal to the number of UPC threads, and if it is not `upc_forall` loop.

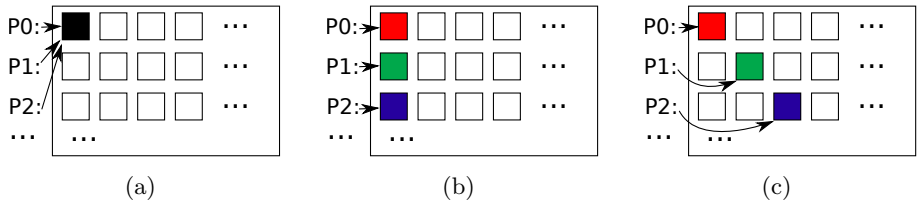

Figure 6.1: Different schemes of accessing a shared array. The shared object is allocating in blocked form. Each row represents data residing in one UPC thread and each box is an array element. The different access types are: (a) baseline: all UPC threads access the same data; (b) 'Skewed': each UPC thread access elements from a different UPC thread; (c) 'Skewed plus': each UPC thread access elements from a different thread and from a different point inside the block.

The compiler categorizes the loops in two categories based on the loop upper bound and shared access type. The compiler applies the transformation depending on the loop category. Figure 6.2 presents the compiler algorithm. The compiler makes the decision based on the usage of `upc_memget` and `upc_memput` calls, and the value of loop upper bound. This type of code is more likely to contain all-to-all communication, thus making it ideal target for the random shuffled solution. The compiler categorizes the loops in:

- Loops that have coarse-grained transfers and whose upper bound is the number of UPC threads. In this case, the runtime returns to the program a look-up table with the number of UPC threads equal to the array size. The contents of the look-up table are the randomly shuffled values for the induction variable. We use this technique only to loops with the upper bound equal with the number of the threads. Thus, the range of shuffled values range from 0 up to `THREADS-1`. To improve the performance of this approach, the runtime creates the shuffled array at the initialization phase. Next, the compiler replaces the induction variable inside the body of the loop with the return value of the look up table. Listing 6.2 presents the final form of the transformed loop. The all-to-all communication pattern belongs to this category.

- Loops that contain fine-grained communication or contain coarse-grained are transfered but in these cases the upper bound of the loop is different from the number of UPC threads. In this case, the compiler skews the iterations in such a way that each UPC thread starts executing from a different point in the shared array. The compiler uses the simple form to 'skew' the iterations to avoid the creation of additional runtime calls. Finally, the compiler replaces the occurrences of the induction variable inside the loop body. Listing 6.3 illustrates the final form of a loop containing fine-grained communication.

```
1  shared [N/THREADS] double X[N];
2  shared [N/THREADS] double Y[N];
3
4  void memget_threads_rand(){
5     uint64_t i=0, block = N/THREADS;
6     double *lptr = (double *) &X[block*MYTHREAD];
7     uint64_t *tshuffle = __random_thread_array();
8     for (i=0;i<THREADS;i++){
9        uint64_t idx = tshuffle[i];
10       upc_memget( lptr, &Y[idx*block], block*sizeof(double));
11    }
12 }
```

Listing 6.2: Compiler transformed loop with coarse-grained access.

```
1  shared [N/THREADS] double X[N];
2
3  void memget_threads_rand(){
4     uint64_t i=0, block = N/THREADS;
5
6     for (i=0;i<THREADS;i++){
7        uint64_t idx = ( i + MYTHREAD*block ) % N;
8        ... = X[idx];
9     }
10 }
```

Listing 6.3: Compiler transformed loop with fine-grained access. We assume the upper bound to be equal with the size of the array

When the compiler selects the simple form to skew the loop iterations the upper bound of loop is not equal to the number of threads.

6.2 Experimental results

This experimental evaluation assesses the effectiveness of the loop transformation. This assessment includes (1) the performance on microbenchmarks using manual code modifications on a large number of UPC threads. This helps to understand the maximum speedup that can be achieved and the potential performance bottlenecks. Second this section presents (2) the performance of compiler transformed microbenchmarks and real applications.

6.2.1 Methodology

The evaluation uses two different Power 775 machines. The first one has 1024 nodes and is used to evaluate the manual code modifications. The second system contains 64 nodes. It allows runs with up to 2048 UPC threads and it is used to evaluate the automatic compiler transformations.

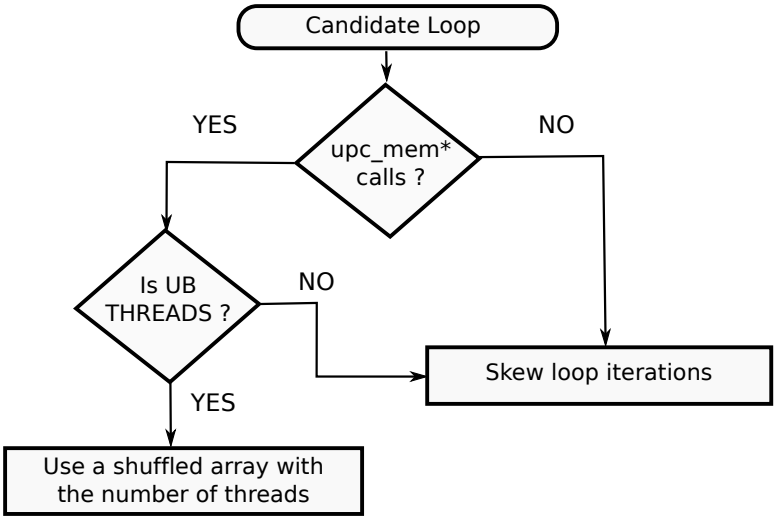

Figure 6.2: Automatic compiler loop scheduling.

We use one process per UPC thread and schedule one UPC thread per Power7 core. The UPC threads are grouped in blocks of 32 per node, and each UPC thread is bound to its own core. The experimental evaluation runs each benchmark five times. The results presented in this evaluation are the average of the execution time for the five runs. All benchmarks are compiled using the '`-qarch=pwr7 -qtune=pwr7 -O3 -qprefetch=aggressive`' compiler flags and with dynamic number of UPC threads. The baseline versions include all the optimizations presented in Chapter 4 and Chapter 5, although they don't optimize benchmarks that contain coarse-grained data transfers. The evaluation varies the size of the data set with the number of UPC threads (weak scaling).

Benchmarks and Datasets

The evaluation uses three microbenchmarks and four applications. The microbenchmark is a loop that accesses a shared array of structures. There are three variations of this microbenchmark. In all versions, each UPC thread executes the same code in the loop. In the **upc_memput** microbenchmark, the loop contains coarse-grained `upc_memput` calls. Listing 6.4 presents the code used in `upc_memput` benchmarks. The **fine-grained get** contains shared reads and the **fine-grained put** contains shared writes. The evaluation also uses a number of benchmarks: (i) the Sobel 9-point stencil benchmark [50]; (ii) Gravitational N-Body Fish [49]; (iii) Bucket-sort [76]; and (iv) NAS FT [77]. Section 3.2.2 provides detailed information about the applications.

6.2.2 Limit study

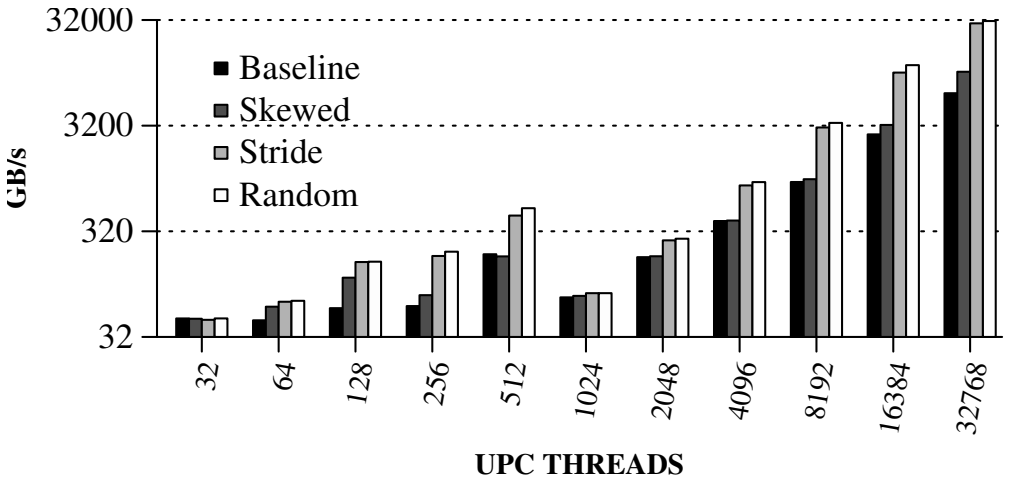

Figure 6.3: Effect of loop scheduling policies on performance for upc_memput.

```
1  #define N (1llu<<32)
2  shared [N/THREADS] double X[N];
3  shared [N/THREADS] double Y[N];
4
5  void memget_threads(){
6    uint64_t i=0; uint64_t block = N/THREADS;
7    double *lptr = (double *) &X[block*MYTHREAD];
8
9    for (i=0;i<THREADS;i++){
10     uint64_t idx = ( i );
11     upc_memput( &Y[idx*block], lptr, block*sizeof(double));
12   }
13 }
```

Listing 6.4: Example of upc_memget loop in UPC.

Figures 6.3 presents the results for the coarse-grained microbenchmark. The drop of the performance when going from 512 to 1024 UPC threads is due to the HUB link limits. The microbenchmarks use two supernodes when running with 1024 UPC threads. Thus, a portion of communication between the UPC threads uses the (remote) D-Links. Furthermore, the *strided* version has lower performance than the *random* shuffled version. The traffic randomization balances the use of global links reducing contention. In contrast, the *strided* version creates less randomized traffic and created predicted traffic by using different node on each loop iteration. Overall, performance for a coarse-grained communication pattern is better when using random allocation. Note, that the is no *skewed plus* version because the *upc_memget* and *upc_memput* calls always start from the beginning of the block.

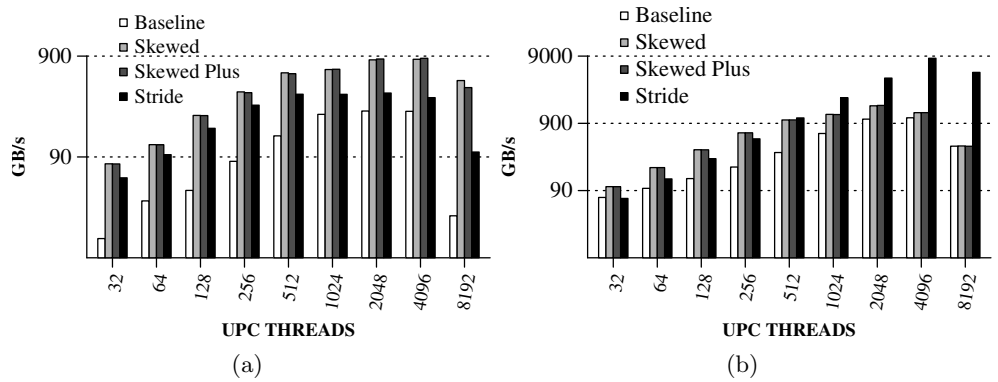

Figure 6.4: Effect of loop scheduling policies on performance for fine-grained get (b) and fine-grained put (c).

Figure 6.4 illustrates the performance of the fine-grained microbenchmarks. The *skewed* and *skewed plus* versions have similar performance, in fine-grained category. Thus, using a different starting point inside the block has no real impact in the performance. The *strided* version has worse performance than the *skewed* version in fine-grained get version. On the other hand, the *strided* has better performance using the fine-grained put version. This occurs because the runtime issues in-order remote stores that target the same remote node.

Moreover, the performance of the fine-grained put is an order of magnitude faster than the fine-grained get. This behaviour is noticeable especially in the strided version. The main reason behind this is that the runtime allows overlapping of store/put operations when the destination node is different. On the other hand, in the read/get operations, the runtime blocks and waits the transfers to finish.

Overall, loops that contain coarse-grained `memget/memput` transfers, the compiler should use random shuffle. On the other hand, the compiler should use skewing transformation for loops with fine-grained communication.

6.2.3 Compiler-assisted loop transformation

This section compares the automatic compiler transformation with the manual transformation. The main difference is that the manual approach avoids additional overhead by inserting runtime calls. Figure 6.5 compares the compiler-transformed and hand-optimized code. While the performance of the manual and compiler-transformed fine-grained microbenchmarks is similar, the compiler transformation achieves slightly lower performance than the hand-optimized benchmark precisely because of the insertion of runtime calls.

There are three different categories of performance patterns in the applications. Figure 6.6 presents the application results. The first category includes applications that have performance gain compared with the version without the scheduling (baseline), such as the NAS FT benchmark. This benchmark achieves from 3%

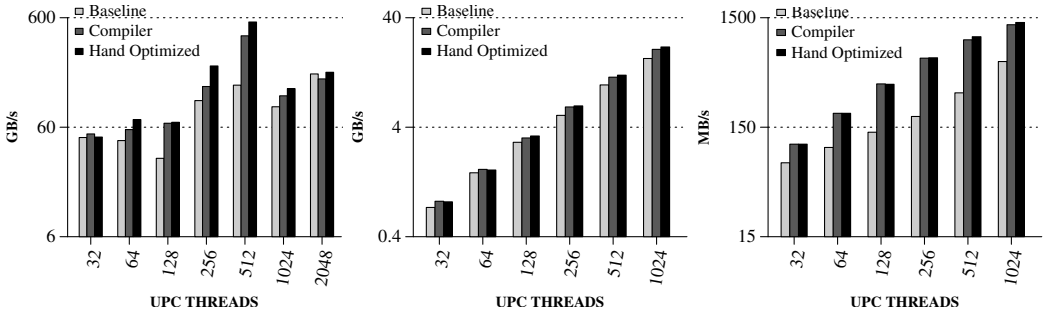

Figure 6.5: Comparison of compiler-transformed and hand-optimized code: upc_memput (a), fine-grained get (b), and fine-grained put (c).

up to 15% performance gain, due to its all-to-all communication pattern. The second category contains applications that have minimal performance gains, such as the Gravitational Fish benchmark. The third category contains benchmarks that exhibit performance decrease, such as the Sobel benchmark. The performance of the Sobel benchmark decreases up to 20% compared with the baseline version, because of poor cache locality. Table 6.1 presents the cache misses for different cache levels in the case of the Sobel benchmark, using the hardware counters.

Cache level	Cache Miss Ratio Baseline (%)	Cache Miss Ratio Scheduling (%)
Level 1	0.14%	0.19%
Level 2	0.19%	24.49%
Level 3	0.32%	28.84%

Table 6.1: Local cache miss ratio using 256 Cores for Sobel benchmark using 256 UPC threads. Results are the average from each of 256 cores. Local miss ratio is calculated by diving misses in this cache the total number of memory accesses to this level of cache.

There are minor differences between the baseline and the transformed version of bucket-sort when using the benchmarks with enabled the local sort. Figure 6.7 presents the results for bucket-sort with enabled and disabled local sort. However, the transformed version has better results — up to 25% performance gain — than the baseline version when only the communication part is used. In Figure 6.7(b) the performance for 32 UPC threads is 40% lower than the baseline because of the overhead of the additional runtime calls. The effectiveness of the loop transformation is limited when running less than 32 UPC threads.

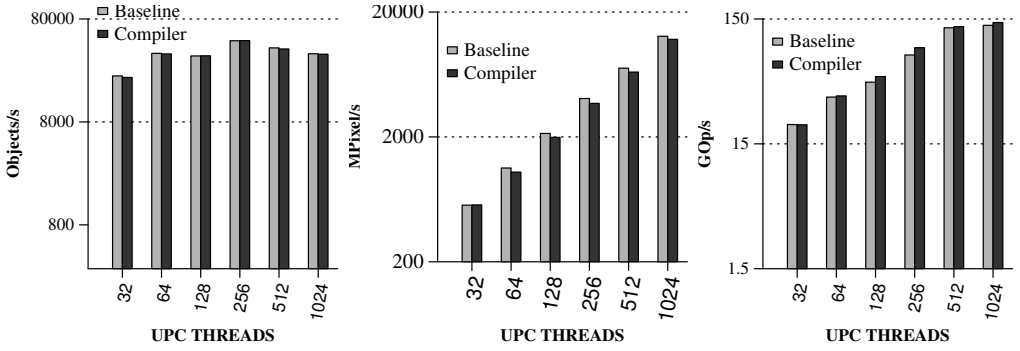

Figure 6.6: Comparison of baseline and compiler-transformed code for fish (a), Sobel (b), and NAS FT (c).

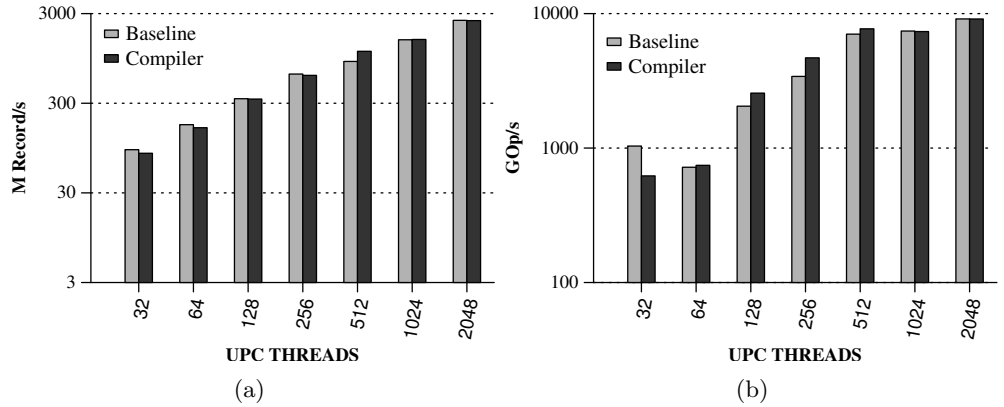

Figure 6.7: Comparison of baseline and compiler-transformed code for bucket-sort (a) and bucket-sort with only the communication pattern (b).

6.3 Chapter Summary and Discussion

This chapter examines the problem of concurrent access of shared data from different UPC threads and proposes different approaches solving the deficiency. Manual and compiler transformations are presented and evaluated for their effectiveness. The evaluation indicates that the compiler transformation is an effective technique for increasing the network performance in UPC languages. The results show a performance gain raging from 5% up to 25% compared to the baseline versions of the applications. Moreover, microbenchmark results show even higher performance gains of up to 3X. On the other hand, the loop transformation has a negative effect on the cache locality in some benchmarks, such as Sobel. Nevertheless, the compiler transformation is useful for a number of codes in order to improve performance in languages like UPC, which can create bottlenecks by the the programmer.

Chapter 7

Related Work

There has been extensive research in the area of parallel programming languages and the optimization of fine-grained access. This chapter provides an overview of the optimizations for improving the performance of the PGAS languages, made available in the literature to date. Specifically, Section 7.1 presents the related work for prefetching and Section 7.2 reports on the related work for inspector-executor. Section 7.3 and Section 7.4 look at the compile time transformations and the runtime optimizations, respectively. Section 7.5 presents the related work for loop scheduling. Proposed language extensions to improve locality are presented in Section 7.6. Section 7.7 describes application specific optimizations. Finally, Section 7.8 presents representations that can describe array accesses.

7.1 Prefetching

Compiler transformation for reducing memory latency using the software prefetching with the inspector-executor approach, have been long investigated [91, 92]. A popular alternative shared memory programming model is the Software Distributed Memory Systems (DSMs), such as Nanos DSM [4], ThreadMarks [5] and ParaADE [6]. Similarly to the PGAS programming model, the programmer sees a global address space and the runtime is responsible for the communication. For example, TreadMarks uses a runtime system to detect accesses to shared data and to manages interprocess communication. These implementations include a form of software prefetching. However, most of the optimized DSM systems are based on the page fault mechanism with page prefetching and often have poor performance on fine-grained communication [7].

7.2 Inspector-executor approaches

The inspector-executor strategy is a well know optimization technique for global name space programs for distributed execution. Inspector loops were first introduced to extract parallelism from loops with irregular data accesses [37] and later

for the global name space programming model on distributed architectures [23, 82]. The inspector loop analyzes the communication pattern and the executor loop performs the actual communication based on the results of the analysis carried out by the inspector loop.

The Vienna Fortran Compilation System framework [24] implements techniques similar to ours. The compiler creates sets of inspector and executor loops for the High Performance Fortran `FORALL` loop construct. The inspector loop analyzes the communication pattern of the application and the executor loop performs the actual communication based on the results of the analysis performed in the inspector loop. In contrast, we apply the strip-mining transformation on the original loop to achieve overlapping of communication with unrelated computation in the loop, and our analysis is not limited to forall loops. Finally, their algorithm requires the globalization and privatization of arrays referenced in the forall loop. The algorithm requires that the local part of the array is copied to shared memory and modified parts of this copy be copied back to the process private memory. In contrast, our approach does not require neither of these steps.

Titanium language [14] use the inspector-executor execution model is the runtime and compiler support for irregular computation. This work targets the built-in foreach loops in Titanium language [26]. This is a major limitation which our approach overcomes. In addition they do not apply any further loop transformation, such as to strip mine the loop. However, this approach causes the inspector loop to capture every shared access present in the parallel loop and it will create much higher runtime overhead. Also, the profitability of their loop transformation in this paper is estimated at compile time. This is an unrefined approach that will result in a program optimized for a specific architecture.

Other researchers use the inspector-executor approach for High Performance Fortran (HPF) [25], but stumble upon the limitation of no overlapping communication with computation and they don't limit the number of inspect elements. Authors in [82] present general techniques for caching, data coalescing, and message vectorization. The authors explore the potential benefit of these techniques with irregular applications.

7.3 Compile-time Optimizations

Compile time optimizations focus on changing the provided source code to improve the performance with limited or not at all help from the compiler. The optimizations focus on simplification of the generated code, privatization of shared pointer in `upc_forall` loops, coalescing of shared objects, and automatic overlap of communication and computation. The most relevant drawback of this approach lies in that many compiler optimizations are based on the available information at compile time. For example, the programmer that wants to develop portable code, avoids specifying information at compile time, such as the number of nodes, threads, and blocking factor. In this case a number of optimizations, such as the

replacement of shared pointer with simple pointer (privatization) will fail because the data distribution is not known at compile time.

7.3.1 Code simplification

The most common optimization is the `upc_forall` loop simplification [32, 93]. The compiler optimizes the parallel loop by eliminating the branch inside the loop, created by the affinity expression. For example, the parallel loop:

```
upc_forall ( i =0; i<N; i ++; i ){
    a[i] = b[i] + scalar * c[i]
}
```

will be transformed to:

```
for ( i =0; i < N; i ++){
    if (( i % THREADS) == MYTHREAD )
        a[i] = b[i] + scalar * c[i] ;
}
```

However after the optimization final loop will not contain the affinity branch:

```
for ( i =MYTHREAD; i<N; i +=THREADS){
    a[i] = b[i] + scalar * c[i] ;
}
```

Another optimization that the compiler can apply is the elimination of pointer arithmetic on shared accesses [22]. Shared pointer arithmetic is expensive because a runtime call is automatically created upon each access. When the compiler replaces this runtime call with a simple pointer dereference the compiler can use common subexpression elimination to avoid the creation of runtime calls. Table 7.1 illustrates an example of the optimization. Before the optimization the compiler will replace each **(p+j)* expression with runtime calls. After the optimization the compiler will insert only one call before the usage of **(t)* and it will replace the other dereferences.

7.3.2 Shared-Pointer Privatization

The compiler uses information provided by the affinity expression of an `upc_forall` loop to privatize the shared accesses that are to the local partition of the memory [32, 33]. For instance, an affinity expression that is pointer-to-shared usually indicates that the references are local memory. Thus, the compiler can transform the *fat* shared pointer into a *thin* private pointer and save a runtime call. Unfortunately, this approach only works when the `upc_forall` construct is used or requires that the physical data placement be known at compile time.

Before optimization	After optimization

`shared int *p = foo();`
`int i = ...`
 	
`int j = f( i, ... );`
 	
`... = *(p+j);`
`... = *(p+j);`
	`shared int *p = foo();`
	`int i = ...`
	
	`int j = f( i, ... );`
	`shared int *t = p + j`
	
	`... = *(t);`
	`... = *(t);`

Table 7.1: Example using the pointer arithmetic optimization.

Before optimization	After optimization

`struct S {int x; int y;};`
 	
`shared struct S *p;`
 	
`... = __ptr_deref(&p->x, 4);`
`... = __ptr_deref(&p->y, 4);`
	`struct S {int x; int y;};`
	
	`shared struct S *p;`
	
	`__ptr_deref_co(&t, p, 8)`
	`...= t[0];`
	`...= t[1];`

Table 7.2: Example using the coalescing optimization.

7.3.3 Shared Object Coalescing

Optimizations for data coalescing using static analysis exists in Unified Parallel C [29, 22] and High Performance Fortran [30, 31]. Table 7.2 presents an example of the optimization. A compiler uses data and control flow analysis to identify shared accesses to specific threads and creates one runtime call for accessing the data from the same thread. However, the existing locality analysis algorithm is not applicable to other constructs, such as a for loops or do/while loops. Furthermore, it requires information for the physical data placement at compile time, even for *upc_forall* loops, to optimize the shared accesses. Finally, in practice many existing UPC programs do not make extensive use of *upc_forall* loop constructs, discounting a substantial optimization opportunity. In contrast, we provide a generic approach to coalesce data accesses at runtime that overcomes the necessity for prior knowledge of physical data mapping and without the necessity to use the *upc_forall* loop structure. Other Fortran-like programming models also support static coalescing and are described as message vectorization [94, 82, 95]. The X10 [27] and Chapel [28] languages also support aggregation at compile time to improve the performance.

Before optimization	After optimization

`void foo ( shared int *p){`
 	
`  m = ...`
`  __ptr_assign(&m, ...);`
 	
`  __ptr_deref(&t, ...);`
`  ... = t;`
 	
`}`
	`void foo ( shared int *p){`
	
	`  h1 = __ptr_deref(&t, ...);`
	
	`  m = ...`
	`  h2 = __ptr_assign(&m, ...);`
	
	`  __wait(h1);`
	`  ... = t;`
	
	`  wait(h2);`
	`}`

Table 7.3: Example using the splitting optimization.

7.3.4 Overlapping of communication and computation

Another approach to minimize the communication latency in the PGAS programming model is the split of the issuing shared accesses and the synchronization points. In literature, this approach is referred to either as "split-phase communication" [22] or "scheduling" [32, 96, 31]. The compiler inserts runtime calls to fetch the data as early possible inside the code and inserts a synchronization call just before the use of the data. Other approaches delay the completion of a put operation until either a synchronization point is reached or a procedure exists over the data [97]. Table 7.3 presents an example of the optimization. However, these approaches have limited opportunities for optimizing loop structures, due to the complexity involved in data flow analysis. In contrast, we focus on optimizing fine-grained shared accesses inside loops.

Researchers have also proposed techniques for overlapping computation with communication techniques using a more sophisticated runtime. One approach to exploiting computation and communication overlapping, is to decompose the coarse-grained transfer segments, into strips and to store a handle for each strip [98]. Next, the runtime checks on each access, if the data have been arrived for the current accessing strip. This optimization uses a combination of strip-mining and message scheduling. They aim is to optimize coarse-grained data transfers by using a pipeline approach. The runtime uses the `mprotect` call for each page associated with the strip. The program will continue the execution until the point the program accesses the shared data. At this point, the program receives a `SIGSEGV` signal and executes the associated signal handler. The handler identifies that the faulting address belongs to a transfer buffer, restores access rights and blocks the execution until the transfer is completed. The same algorithm is repeated when a new strip is accessed. The shortcoming of this implementation is the increased overhead due to signal handler and the incompatibility with high speed network controllers.

Optimizing the `upc_memget` runtime calls introduces a more complicated mechanism and dynamic access analysis. The runtime analyzes the accesses between two synchronization statements. If the program enters again in the same code region the runtime tries to prefetch data. When the execution of the region between two synchronization points finishes, the runtime creates a list of possible prefetches and issues them after the exit of synchronization point to overlap communication with computation. The optimization in cooperation with the data aggregation improves significant the performance [97]. In this case, The runtime tries to aggregate the shared access to more coarse-grained messages, to increase the efficiency of the network. One aspect of this approach is the trade-off between aggregated and pipelining messages. However, this approach has the drawback of increased overhead of the runtime, due to access analysis and aggregation analysis.

Authors in [99] present an optimization framework that replaces common fine-grained communication patterns with coarse-grained runtime calls. There runtime calls can be either point to point or collective. The XL UPC compiler [52] is able to detect common initialization idioms and substitute the fine-grained communication in such loops with coarser-grained communication. Examples of these patterns include loops that simply copy all elements of a shared array into a local array, or vice versa, or loops used to set all elements of a shared array with an initial value. The compiler classifies the operations based on the access pattern and replaces the individual shared array accesses with calls to one of the UPC string handling functions: `upc_memget`, `upc_memset`, `upc_memcpy`, or `upc_memput`.

7.4 Runtime optimizations

Runtime optimizations provide a way of reducing the impact of the network latency without altering the source code of the program. This approach has the advantage in case where the source code is not available for re-compilation.

7.4.1 Software caching

The MuPC [100, 34] and HP UPC [101, 102] runtime systems implement software caching. The HP UPC runtime caches the operations on remote nodes and prefetches nearby remote addresses. A four-way set associative cache method is used. The cache block size is controlled by an environment variable. Finally, the caching is disabled in SMP environments. The MuPC cache is a non-coherent, direct mapped, write back cache. Each UPC thread maintains a non-coherent cache for remote scalar references made by a thread. The total cache size scales with the number of threads and each cache block holds cache lines only from the corresponding remote thread. The cache line size is configurable with an environment variable. Caching greatly reduces the memory latency in the relaxed mode by reducing the number of messages.

In other approaches, that use a distributed shared variable directory (SVD), an address cache is implemented. The caching of remote addresses reduces the shared

access overhead and allows better overlap of communication and computation, by avoiding the SVD remote access [35, 54]. Also, a software-cache mechanism creates additional network traffic due to cache coherence.

Finally, the idea of shadowing or overlapping data [103] assumes that references are in a contiguous space around the local data of a distributed array, which normally belongs to a remote node.

7.4.2 Hybrid environments

Various optimizations have been applied to exploit hybrid environments. Most of the work is focused on optimizations for multicore nodes [104], optimizing collective communication on hybrid environments [105], over-subscription [106], and load balancing [107] when using over-subscription.

Over-subscription [106] can improve the throughput of the benchmarks by up to 27%. However, over-subscription is used when running multiple different competitive benchmarks on the same node. Another problematic aspect of this approach is that the default Linux process scheduler decreases the performance when the over-subscription is used. Furthermore, the proper scheduling of the tasks to achieve load balancing is an active research topic [107].

Researchers proposed a number of optimizations in hybrid environments for the collective calls [105, 55]. These optimizations exploit the heterogeneous environment to minimize the overhead of data transfers. For example in the broadcast, the task first exchange data inside the node through inter-process communication or shared memory. Then, only one task will communicate with one of the tasks on each node to propagate the data. Finally, the task responsible for receiving the data, broadcast them inside the node.

Selection between different ways of transmitting data in the collectives of UPC has been also investigated [104]. For example, the runtime can use flat algorithms (the UPC threads communications with all other UPC threads) for small messages, or tree algorithms for large messages. However, these approaches depend on the characteristics of the networks and the machines.

7.5 Loop Scheduling

Overall, memory optimizations is a widely researched topic [108, 109]. Traditional loop transformations focus on increasing the cache performance and bandwidth [110, 111]. Researchers also use loop scheduling techniques to improve the performance of Non-Uniform Memory Access (NUMA) machines [112, 113] and heterogeneous machines [114, 115].

Recent efforts on loop transformations focus on reducing the memory bank conflicts [88, 89], especially in GPUs [116] and embedded systems [117]. Similarly, our approach uses loop transformation by distributing the shared access across the UPC threads aiming at increasing the network efficiency. Unlike the majority of

the previous works, our approach employs a random distribution to improve the performance.

The Dragonfly interconnect [70] uses randomized routing as an effective approach to reduce hotspots. What is more, this approach requires additional hardware support. Instead we advocate moving the complexity to software. Our compiler transformation can offer comparable performance by randomizing the source and destination pairs involved in communication.

Other approaches to increasing the similarity of a uniform random traffic pattern include randomized task placement [118] and adaptive routing [90]. The authors in [90] randomize the routing at runtime to achieve better performance based on different patterns. Randomized task placement [118] can increase the amount of randomized traffic and avoid traffic. However other researchers proved [119] that despite the improved results, the performance of randomized uniform traffic is still far from ideal.

7.6 Language Extensions

Multi-blocking allocation of shared data in UPC [120] and HPF+ Fortran [121] is a common technique used to minimize the number of messages exchanged. However, these approaches are not part of the language standard and they still incur overhead from a library call that is automatically creation. Some other proposals made in literature are the following: First, the HPF+ Fortran [121] to achieve better performance and address the limitations of the High Performance Fortran. The proposed extensions include indirect and multiblock allocation. The arrays are distributed dynamically depending on the phase of the algorithm — in contrast with the standard blocking allocation. Furthermore, the Researchers also propose dynamic data distribution for adaptive memory allocations. The arrays are distributed dynamically depending on the phase of the algorithm, in contrast with the standard blocking allocation. The language also contains some extensions for specifying that the data are local. Finally, the language supports shadow (for regular) or "Halo" regions to specify the remote accesses. Second, researchers have proposed some language extensions to make the Fortran 90 programs run in parallel [122]. The main contribution is that the arrays are distributed among different threads. The framework contains one pre-processor that transforms Fortran 90 to Fortran 77 and one runtime for communication.

7.7 Application specific optimizations

Researchers have also proposed application specific optimizations that focus on specific platforms. Unfortunately, these approaches are hard to generalize and are application domain specific. Nevertheless, researchers should always explore these efforts when optimizing similar applications.

An example of the above is the optimization of stencil computation in High Performance Fortran [123] using communication unioning [124]. There are also compilers that optimize stencil computation for specific machine architectures such as the convolution compiler [125]. The compiler uses the polyshift [126] communication mechanism for CM-2 machine. The XL HPF compiler of IBM [127] utilizes similar approach to decrease the number of messages.

Another example are code improvements for large-scale graph analysis. Programmers can easily map the shared-memory graph algorithms to PGAS environments, although it is unlikely that naive implementations achieve high performance for large datasets. Code improvements for increasing their performance includes message coalescing and privatization in the context of distributed graph algorithms [128].

Other researchers proposed algorithms for the HPCC RandomAccess Benchmark to increase the efficiency [129]. However, the authors focus the approaches to optimize communication for the Red Storm Machine. The researchers use different approaches to allocation and communication to exploit the hypercube architecture for aggregating the messages.

7.8 Array Access Analysis

Linear Memory Access Descriptors (LMAD) [40, 41] is a well known representation that describes the accesses on a array. This representation is used for array accesses analysis, coalescing of accesses, and for privatizing the array accesses on various platforms. For instance, Xhu et al. use the LMAD representation to translate program manually for distributed shared memory (DSM) systems [130]. Xunhao Li [131, 132] and Garg et al. [39] use a subset of LMAD, Restricted Constant Strided Linear Memory Access Descriptor(RCSLMAD) to identify memory locations of accessed array elements in Graphic Processor Units (GPUs).

Other alternative representations of arrays accesses is the Systems of Affine Recurrence Equations (SAREs) over polyhedral domains [133]. Researchers created various tools to automatize the privatization of the data in GPUs environments [134, 135, 136].

Chapter 8

Conclusions and Future Work

This dissertation presented optimizations that increase the performance of UPC applications containing fine-grained communication. In order to efficiently improve the performance of this latter type of programs, the programmer or the compiler framework must overcome three challenges: (i) the low network efficiency due to the small messages; (ii) the large number of runtime calls; and (iii) network hotspot creation for the non-uniform distribution of network communication, especially in all-to-all patterns.

To address these deficiencies, this thesis suggested optimizations, which can be grouped under three major categories. First, this thesis proposes the utilization of an improved inspector-executor transformation (Chapter 4) [36] to solve the low network efficiency. The compiler transforms the loop in the inspector-executor form, and then it stripe-mines the loops iterations into smaller blocks to achieve overlap of computation and communication. Between the inspector and executor loop the runtime analyzes and aggregates the remote accesses into bigger messages.

Second, the thesis proposed optimizations to decrease the number of runtime calls in the inspector-executor transformation (Chapter 5). The optimizations employ a number of techniques to successfully remove the calls. Namely, the use of Constant Stride Linear Memory Address Descriptors (CSLMADs) [40] with regular accesses; the use of a temporary array to collect shared indexes for irregular accesses; static coalescing when using structs [38]; a lightweight loop code motion designed for the UPC language. The evaluation demonstrated how eliminating unnecessary runtime calls inserted by the compiler when accessing shared data is important to achieve good performance.

The final part of the thesis presented optimizations to decrease the possibility of hotspot creation. UPC language is subject to hotspot creation or oversubscription of the node due to the transparency allowed to the programmer. To effectively avoid network congestion, the programmer or the runtime must spread non-uniform traffic evenly over the different links. Chapter 6 explored the possible approaches of distributing the accesses through the network: (i) by manually modifying the source code; (ii) by using compiler assistance loop transformations

to automatically improve performance.

Based on these results, we draw the following conclusions:

- Unified Parallel C programming model can provide MPI-comparable performance in applications that contain fine-grained regular accesses, combined with the optimizations presented.
- To provide performance comparable to coarse-grained application, it is important for the compiler to eliminate unnecessary runtime calls. Simple approaches, such as the inspector-executor transformation, are not enough.
- For applications with irregular shared references, the programmer benefits by expressing the accesses in the form of read-modify-write of data. Alternatively, the compiler optimizations, as presented in this thesis, can provide an order of magnitude performance gain, but there is still room to close the gap with the manual optimized benchmarks.

8.1 Publications

During the course of the thesis, we have published a number of articles. First of all, the algorithm of inspector-executor presented in Chapter 4 was submitted as patent [43]. The patent describes the basic design aspects of the transformation.

> **Michail Alvanos**, Ettore Tiotto. Software prefetching technique for Partion Global address Space (PGAS) languages. Canada Patent Application No. 2762563 - US Patent Application US-20130167130.

After filling the patent, IBM gave permission to publish the transformation and the initial results in the CASCON conference [36]. This work explains in more detail the implementation of the patent and provides an evaluation up to 256 Cores. Following this presentation, we received feedback from experts in the field. Such insights were important for developing this work further.

> **Michail Alvanos**, Montse Farreras, Ettore Tiotto, and Xavier Martorell. Automatic Communication Coalescing for Irregular Computations in UPC language. *In Conference of the Center for Advanced Studies, CASCON 2013.*

Our next step was to address the comments made on the previous publication, scale up the experiments up to 32768 cores, and provide incremental optimizations [38]. The contribution of this article is the combination of compile time (static), as presented in Chapter 5, and runtime (dynamic) coalescing techniques that do not require knowledge of physical data mapping. The evaluation yielded speedups from 1.15X up to 21X compared to the baseline unoptimized versions, achieving up to 63% the performance of the MPI versions.

> **Michail Alvanos**, Montse Farreras, Ettore Tiotto, Jose Nelson Amaral, and Xavier Martorell. Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC. In *International Conference on Supercomputing, ICS 2013.*

During the development of the International Conference on Supercomputing paper, we came across an interesting outcome: The network was performing poorly in a number of benchmarks. The problem was that the traffic was non-uniform, creating hotspots on the network. Thus, our initial reaction was to present manual and automatic compiler transformations to distribute the traffic [44]. Despite the limited evaluation, the short article offers a glimpse of the UPC language pitfalls to the community and provides possible solutions.

> **Michail Alvanos**, Gabriel Tanase, Montse Farreras, Jose Nelson Amaral, Xavier Martorell. Improving performance of all-to-all communication using better loop scheduling in PGAS environments, In *International Conference on Supercomputing, ICS 2013.* 2 Pages.

The short paper/poster on the International Conference on Supercomputing sparked the exploration of a new set of optimizations, targeting both applications with fine-grained and coarse-grained communication patterns [45]. Thus, the last part of the thesis (Chapter 6) was published as a short paper on the International Conference on PGAS Programming Models:

> **Michail Alvanos**, Gabriel Tanase, Montse Farreras, Jose Nelson Amaral, Xavier Martorell. Improving Communication Through Loop Scheduling in UPC. In *7th International Conference on PGAS Programming Models, PGAS 2013*

Finally, as a member of the XL UPC Toronto compiler team, the author was involved in a technical report that evaluates all the compiler and runtime optimizations evaluated in the Power 775 machine [52]. The report provides insights on the expected performance of UPC applications in the Power 775 machines:

> Gabriel Tanase, Gheorghe Almaśi, Ettore Tiotto, **Michail Alvanos**, Anny Ly, Barnaby Dalton. Performance Analysis of the IBM XL UPC on the PERCS Architecture. *IBM Technical Paper. RC25360, 2013.*

Parts of this work are still under review for publication. For example, the second part of the Chapter 5 is not properly described in the ICS paper [38]. Furthermore, the details of the Chapter 6 are not presented in the current publications [44, 45].

8.2 Productization

The transformation presented in Chapter 4 was productized under the `-qprefetch=aggressive` flag of XL UPC compiler [52, 53, 137] on Power 775 machine [138]. According to TOP500 [139] there are 9 installed supercomputers providing a total of 5361 TFlops of processing power. Installed machines are ranging from 2,000 to over 64,000 Power7 processing cores.

8.3 Future Work

The current prototype implementation can be improved in several ways. First of all, the compiler has strict requirements for the inspector-executor transformation. The other transformations have fewer requirements, but the effectiveness is limited. The loops must be normalized and monotonically increasing to completely remove the calls from the inspector and executor loops. Selecting a simpler inspector-executor transformation without the blocking transformation can bridge these requirements, despite possible drawbacks.

Furthermore, the existence of runtime calls inside the loops decreases the opportunities for optimizing the loop. Inter-procedural analysis can give the additional information about the loop analysis to increase the opportunities for optimizing loops that contain procedure calls. The amount of iterations to analyze and prefetch is different and depends on the access pattern, number of UPC threads, and loop upper bound.

Finally, there are two applications categories that this thesis does not examine. The first category included 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 patterns. Another category is the benchmark that contains sparse matrices. In this scenario, the inspector-executor optimization fetches only the first level of the accesses leaving the blocking communication inside the execution loop.

For example, Listing 8.1 presents an algorithm for connected components in UPC [128]. The question here is how we optimize these accesses ? In the first loop, the compiler privatizes the access in the El array, because it is `upc_forall` structure. Furthermore, the compiler can use the inspector-executor transformation to prefetch the D[u] and D[v] accesses. However, there is an indirect access on D array that uses the D[v] in line 5.

The second loop contains an even more complicated expression that traverses the graph. Note that the loop is irregular and contains indirect accesses. How can the programmer optimize this piece of code? Even more, how the compiler developer teach the compiler to apply optimizations on this irregular loop?

```
1   gr = 0;
2   upc_forall ( i=0; i<m; i ++; &El[i] ){
3     u = El[i].u; v = El[i].v;
4     if ( D[u] < D[v] ) {
5       D[ D[v] ] = D[u] ;
6       gr = 1;
7     }
8   }
9
10  gr = all_reduce_i ( gr , UPC ADD) ;
11
12  if ( gr == 0) break ;
13
14  upc_forall ( i=0; i<n; i++; &D[i]){
15    while (D[i] != D[ D[i] ] )
16      D[i] = D[ D[i] ] ;
17  }
```

Listing 8.1: UPC algorithm of connected components graph.

8.4 Survival of the UPC language

High Performance Fortran (HPF) [140] was a promising programming model for high performance computing. HPF has the same goals with the Unified Parallel C language: provide a programming model for scalable parallel systems, according to which (i) the programmer sees a single shared address space, (ii) communication is implicitly generated, (iii) the performance is comparable to the MPI hand tuned version.

Unfortunately, the research and development process of the HPF compilers and runtime stopped some years later. Kennedy explained why HPF did not survive [141]. The main reasons for the limited adoption of the language are: (i) the fact that programmers usually rewrite the application for the specific compiler and machine; (ii) the use of immature compiler technology that leads to poor performance; (iii) missing tools.

Despite the adequate number of UPC compilers [62, 142, 52, 143], not all of them follow the standards or have good performance. For example, many of the advanced optimizations of XL UPC compiler are not yet supported by the Cray UPC compiler [143]. Another example is the gcc UPC compiler. In this case, the programmer must specify the number of threads when declaring shared arrays on the heap. In a typical scenario, the programmer will have to put some time and effort into porting the applications to a different compiler and optimizing it to achieve the maximum performance.

Despite all these challenges that HPC community and compiler developers must face, the UPC language has greater opportunities than the HPF. The main reason is the support from the open community with the availability of the Berkeley UPC compiler [142] and the gcc UPC Compiler [62]. Both projects are open source and available to download. Finally, they include support for many architectures and Interconnects, partially solving the portability issue.

Bibliography

[1] L. Dagum and R. Menon, "Openmp: An industry standard api for shared-memory programming," in *IEEE International Conference on Computational Science and Engineering*, 1998.

[2] 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.

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

[4] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta, "Running OpenMP applications efficiently on an everything-shared SDSM," in *In Proc. of IPDPS 04*, pp. 35–42, 2004.

[5] 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.

[6] Y.-S. Kee, J.-S. Kim, and S. Ha, "Parade: An openmp programming environment for smp cluster systems," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, (New York, NY, USA), pp. 6–, ACM, 2003.

[7] 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*, OSDI '99, (Berkeley, CA, USA), pp. 215–228, USENIX Association, 1999.

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," in *Proceedings of the Supercomputing '93 Conference*, (Portland, OR), pp. 262–273, November 1993.

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

[10] R. Numwich and J. Reid, "Co-array fortran for parallel programming," tech. rep., 1998.

[11] 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`.

[12] Cray Inc, "Chapel Language Specification Version 0.8," April 2011. `http://chapel.cray.com/spec/spec-0.8.pdf`.

[13] 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, pp. 519–538, Oct. 2005.

[14] 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.

[15] J. Lee and M. Sato, "Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pp. 413–420, 2010.

[16] Francois Cantonnet, Yiyi Yao, Mohamed M. Zahran, and Tarek A. Elghazawi, "Productivity Analysis of the UPC Language," in *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2004.

[17] Rajesh Nishtala and George Almasi, "Performance without pain = productivity: Data layout and collective communication in UPC," in *In Principles and Practices of Parallel Programming (PPoPP)*, 2008.

[18] K. A. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. N. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. L. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pp. 24–32, 2007.

[19] C. Coarfa, Y. Dotsenko, J. M. Crummey, F. Cantonnet, T. E. Ghazawi, A. Mohanti, Y. Yao, and D. C. Miranda, "An evaluation of global address space languages: co-array fortran and unified parallel C," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pp. 36–47, 2005.

[20] Christopher Barton, George Almasi, Montse Farreras, and Jose Nelson Amaral, "A characterization of shared data access patterns in upc programs," in *In Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pp. 111–125, 2006.

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

[22] 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.

[23] C. Koelbel and P. Mehrotra, "Compiling Global Name-Space Parallel Loops for Distributed Execution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 440–451, October 1991.

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

[25] 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*, pp. 706–711, 2002.

[26] 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.

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

[30] 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)*, pp. 14–25, 2005.

[31] 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.

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

[33] W.-Y. Chen, *Optimizing Partitioned Global Address Space Programs for Cluster Architectures*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2007.

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

[35] M. Farreras, G. Almási, C. Cascaval, and T. Cortes, "Scalable rdma performance in pgas languages," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pp. 1–12, IEEE, 2009.

[36] 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*, CASCON '12, ACM, 2012.

[37] 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.

[38] 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.

[39] R. Garg and J. N. Amaral, "Compiling python to a hybrid execution environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 19–30, ACM, 2010.

[40] Y. Paek, J. Hoeflinger, and D. A. Padua, "Efficient and Precise Array Access Analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 1, pp. 65–109, 2002.

[41] Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of array access patterns for compiler optimizations," in *ACM SIGPLAN Notices*, vol. 33, pp. 60–71, ACM, 1998.

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

[43] M. Alvanos and E. Tiotto, "Data Prefetching and Coalescing for Partitioned Global Address Space Languages," Oct. 24 2012. US Patent App. 13/659,048.

[44] M. Alvanos, G. Tanase, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell, "Improving performance of all-to-all communication through loop scheduling in PGAS environments," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 457–458, ACM, 2013.

[45] M. Alvanos, G. Tanase, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell, "Improving Communication Through Loop Scheduling in UPC," in *Proceedings of 7th International Conference on PGAS Programming Models*, 2013.

[46] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[47] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[48] ISO/IEC JTC1 SC22 WG14, "ISO/IEC 9899:TC2 Programming Languages - C," tech. rep., May 2005. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`.

[49] S. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics, Cambridge, U.K.; New York, U.S.A.: Cambridge University Press, 2003.

[50] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: a NPB experimental study," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, (Los Alamitos, CA, USA), pp. 1–26, IEEE Computer Society Press, 2002.

[51] T. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: A NPB Experimental Study," in *In Proceedings of Supercomputing 2002*.

[52] 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. RC25360.

[53] IBM, "XL C and C++ compilers." `http://www-01.ibm.com/software/awdtools/xlcpp/`.

[54] C. Barton, C. Cascaval, G. Almasi, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral, "Shared memory programming for large scale machines," *Sigplan Notices*, vol. 41, 2006.

[55] 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*, ICS '12, (New York, NY, USA), pp. 215–224, ACM, 2012.

[56] G. Shah and C. Bender, "Performance and Experience with LAPI – A New High-Performance Communication Library for the IBM RS/6000 SP," in *Proceedings of the 12th. International Parallel Processing Symposium*, IPPS '98, (Washington, DC, USA), pp. 260–, IEEE Computer Society, 1998.

[57] IBM, *Parallel Environment Runtime Edition for AIX, PAMI Programming Guide, Version 1 Release 1.0, IBM.* 2011. `http://publib.boulder.ibm.com/epubs/pdf/a2322730.pdf`.

[58] M. Farreras and G. Almasi., "Asynchronous PGAS runtime for Myrinet networks.," in *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS)*, Oct 2010.

[59] G. Almási, C. Archer, J. G. Castaños, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen, "Design and implementation of message-passing services for the Blue Gene/L Supercomputer," *IBM Journal of Research and Development*, vol. 49, pp. 393–406, March 2005.

[60] Kumar, Sameer and Dozsa, Gabor and Almasi, Gheorghe and Heidelberger, Philip and Chen, Dong and Giampapa, Mark E. and Blockso Michael and Faraj, Ahmad and Parker, Jeff and Ratterman, Joseph and Smith, Brian and Archer, Charles J., "The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 94–103, ACM, 2008.

[61] D. Bonachea, "Gasnet specification, v1.1," tech. rep., Berkeley, CA, USA, 2002.

[62] "The Berkeley UPC Compiler." http://upc.lbl.gov.

[63] 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.

[64] R. Arroyo, R. Harrington, S. Hartman, and T. Nguyen, "IBM Power7 systems," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 2–1, 2011.

[65] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, "Power7: IBM's Next-Generation Server Processor," *Micro, IEEE*, vol. 30, pp. 7 –15, march-april 2010.

[66] 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.

[67] M. S. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. J. Drake, L. Pesantez, T. Gloekler, J. A. Tierno, P. Bose, and A. Buyuktosunoglu, "Introducing the adaptive energy management features of the power7 chip," *IEEE Micro*, vol. 31, no. 2, pp. 60–75, 2011.

[68] D. J. Kerbyson and K. J. Barker, "Analyzing the performance bottlenecks of the power7-ih network," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, (Washington, DC, USA), pp. 244–252, IEEE Computer Society, 2011.

[69] 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*, SC '11, (New York, NY, USA), pp. 42:1–42:11, ACM, 2011.

[70] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 77–88, IEEE Computer Society, 2008.

[71] T. El-ghazawi and F. Cantonnet, "UPC Performance and Potential: A NPB Experimental Study," in *In Supercomputing2002 (SC2002)*, pp. 1–26, IEEE Computer Society, 2002.

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

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

[74] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) benchmark suite," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[75] H. Kulsrud, R. Sedgewick, P. Smith, and T. Szymanski, "Partition sorting on the CRAY-I," *Institute for Defense Analyses, Princeton, NJ*, vol. 7, p. 78, 1978.

[76] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.

[77] H. Jin, R. Hood, and P. Mehrotra, "A practical study of UPC using the NAS Parallel Benchmarks," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, (New York, NY, USA), pp. 8:1–8:7, ACM, 2009.

[78] S. Muchnick, "Advanced compiler design and implementation," 1997. ISBN 1558603204.

[79] 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*, SC '11, (New York, NY, USA), pp. 42:1–42:11, 2011.

[80] Redbooks, IBM, *IBM Power Systems 775 for AIX and Linux HPC Solution.* 2012. `http://www.redbooks.ibm.com/redbooks/pdfs/sg248003.pdf`.

[81] R. Rajamony, M. W. Stephenson, and W. E. Speight, "The power 775 architecture at scale," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, (New York, NY, USA), pp. 183–192, ACM, 2013.

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

[83] A. C. de Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, 2010.

[84] N. E. Fenton, *Software Metrics: A Rigorous Approach.* Chapman & Hall, Ltd., 1991.

[85] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable hpc system based on a dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 103:1–103:9, IEEE Computer Society Press, 2012.

[86] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, Highly-scalable Dragonfly Topology," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 77–88, 2008.

[87] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[88] Das and Sarkar, "Conflict-free data access of arrays and trees in parallel memory systems," in *Proceedings of the 1994 6th IEEE Symposium on Parallel and Distributed Processing*, SPDP '94, (Washington, DC, USA), pp. 377–384, IEEE Computer Society, 1994.

[89] D. L. Erickson, *Conflict-free access to rectangular subarrays in parallel memory modules*. PhD thesis, Waterloo, Ont., Canada, Canada, 1993. Doctoral Dissertation, UMI Order No. GAXNN-81075.

[90] M. Garcia, E. Vallejo, R. Beivide, M. Odriozola, C. Camarero, M. Valero, G. Rodriguez, J. Labarta, and C. Minkenberg, "On-the-fly Adaptive Routing in High-Radix Hierarchical Networks," in *Parallel Processing (ICPP), 2012 41st International Conference on*, pp. 279–288, IEEE, 2012.

[91] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pp. 62–73, 1992.

[92] S. P. VanderWiel and D. J. Lilja, "When caches aren't enough: Data prefetching techniques," *Computer*, vol. 30, pp. 23–30, July 1997.

[93] W.-Y. Chen, "Building a Source-to-Source UPC-to-C Translator," Tech. Rep. UCB/CSD-04-1369, EECS Department, University of California, Berkeley, Dec 2004.

[94] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing*, vol. 2, no. 2, pp. 151–169, 1988.

[95] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy, "A global communication optimization technique based on data-flow analysis and linear algebra," 1998.

[96] 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*, PACT '04, (Washington, DC, USA), pp. 29–40, IEEE Computer Society, 2004.

[97] 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, 2007.

[98] C. Iancu, P. Husbands, and P. Hargrove, "HUNTing the Overlap," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, (Washington, DC, USA), pp. 279–290, IEEE Computer Society, 2005.

[99] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 361–376, July 1991.

[100] Michigan Technological University., "UPC Projects," 2011. `http://www.upc.mtu.edu`.

[101] Hewlett-Packard, "Compaq UPC compiler," 2011. `http://www.hp.com/go/upc`.

[102] Hewlett-Packard, "Unified Parallel C (UPC) Programmer's Guide," April 2007.

[103] H. Gerndt, "Updating distributed variables in superb," *Concurrency: Practice and Experience*, vol. 2, 1990.

[104] Filip Blagojevic, Paul Hargrove, Costin Iancu, and Katherine Yelick, "Hybrid PGAS Runtime Support for Multicore Nodes," in *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10)*, October 2010.

[105] R. Nishtala and K. A. Yelick, "Optimizing collective communication on multicores," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 2009.

[106] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, "Oversubscription on multicore processors," in *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pp. 1–11, IEEE, 2010.

[107] S. Hofmeyr, C. Iancu, and F. Blagojević, "Load balancing on speed," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, (New York, NY, USA), pp. 147–158, ACM, 2010.

[108] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, pp. 149–206, Apr. 2001.

[109] P. Marchal, J. I. Gómez, and F. Catthoor, "Optimizing the Memory Bandwidth with Loop Fusion," in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pp. 188–193, 2004.

[110] J. R. Allen and K. Kennedy, "Automatic Loop Interchange," in *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, pp. 233–246, ACM, 1984.

[111] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pp. 274–286, 1996.

[112] E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 4, pp. 379–400, 1994.

[113] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on numa multiprocessors," in *Parallel Processing, 1993. ICPP 1993. International Conference on*, vol. 2, pp. 140–147, IEEE, 1993.

[114] M. Cierniak, W. Li, and M. J. Zaki, "Loop scheduling for heterogeneity," in *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*, pp. 78–85, IEEE, 1995.

[115] A. T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu, "A class of loop self-scheduling for heterogeneous clusters," in *Proceedings of the 2001 IEEE international conference on cluster computing*, vol. 291, 2001.

[116] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pp. 225–234, ACM, 2008.

[117] Q. Zhang, Q. Li, Y. Dai, and C.-C. Kuo, "Reducing Memory Bank Conflict for Embedded Multimedia Systems," in *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, vol. 1, pp. 471 –474 Vol.1, June 2004.

[118] A. Bhatele, N. Jain, W. D. Gropp, and L. V. Kale, "Avoiding hot-spots on two-level direct networks," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–11, IEEE, 2011.

[119] A. Jokanovic, B. Prisacari, G. Rodriguez, and C. Minkenberg, "Randomizing task placement does not randomize traffic (enough)," in *Proceedings of the 2013 Interconnection Network Architecture: On-Chip, Multi-Chip*, IMA-OCMC '13, (New York, NY, USA), pp. 9–12, ACM, 2013.

[120] C. Barton, C. Cascaval, G. Almasi, R. Garg, J. N. Amaral, and M. Farreras, "Multidimensional blocking in upc," in *Languages and Compilers for Parallel Computing* (V. Adve, M. J. Garzarán, and P. Petersen, eds.), vol. 5234 of *Lecture Notes in Computer Science*, pp. 47–62, Springer Berlin Heidelberg, 2008.

[121] S. Benkner, G. Lonsdale, and H. P. Zima, "The HPF+ Project: Supporting HPF for Advanced Industrial Applications," in *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, (London, UK, UK), pp. 1155–1165, Springer-Verlag, 1999.

[122] J. H. Merlin, "Adapting fortran 90 array programs for distributed memory architectures," in *Proceedings of the First International ACPC Conference on Parallel Computation*, (London, UK, UK), pp. 184–200, Springer-Verlag, 1992.

[123] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner, "Compiling Stencils in High Performance Fortran," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, Supercomputing '97, (New York, NY, USA), pp. 1–20, ACM, 1997.

[124] G. H. Roth, *Optimizing Fortran90D/HPF for distributed-memory computers*. PhD thesis, Houston, TX, USA, 1997. UMI Order No. GAX97-27596.

[125] W. George, R. G. Brickner, R. G. Brickner, W. George, S. L. Johnsson, S. L. Johnsson, A. Ruttenberg, and A. Ruttenberg, "A Stencil Compiler for the Connection Machine Models CM-2/200," in *In Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, p. pages, 1993.

[126] W. George, R. G. Brickner, and S. L. Johnsson, "POLYSHIFT communications software for the connection machine system CM-200," *Sci. Program.*, vol. 3, pp. 83–99, May 1994.

[127] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo, "An hpf compiler for the ibm sp2," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, (New York, NY, USA), ACM, 1995.

[128] G. Cong, G. Almasi, and V. Saraswat, "Fast PGAS Implementation of Distributed Graph Algorithms," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[129] S. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis, "A simple synchronous distributed-memory algorithm for the hpcc randomaccess benchmark," in *Cluster Computing, 2006 IEEE International Conference on*, pp. 1–7, 2006.

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

[131] J. N. A. Xunhao Li, Rahul Garg, "A new compilation path: From python/numpy to opencl," 2011.

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

[133] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *ACM SIGARCH Computer Architecture News*, vol. 12, pp. 208–214, ACM, 1984.

[134] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, (Juan-les-Pins, France), pp. 7–16, September 2004.

[135] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

[136] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[137] IBM, *IBM XL Unified Parallel C User's Guide*. 2012. Version 12.0.

[138] IBM, "IBM Power 775 server," 2012. `http://www-03.ibm.com/systems/power/hardware/775/`.

[139] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 supercomputer sites," 2013. `http://www.top500.org/`.

[140] "High Performance Fortran language specification, version 1.0. Technical Report ," tech. rep., 1993. CRPC- TR92225.

[141] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of High Performance Fortran: an historical object lesson," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 7–1, ACM, 2007.

[142] "GNU Unified Parallel C (GNU UPC)." `http://www.gccupc.org/`.

[143] Cray, "Cray XE6," 2012. `http://www.cray.com/Assets/PDF/products/xe/CrayXE6Brochure.pdf`.

[144] K. Kennedy and J. R. Allen, "Optimizing compilers for modern architectures: a dependence-based approach," 2001. ISBN 1558602860.

[145] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011. ISBN 012088478X.

[146] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012. ISBN 012383872X.

Appendix A

Terminology

This chapter provides definitions for general compiler terminology that is used throughout the thesis. For additional definitions and a complete view of the compiler terminology, the is encouraged read advanced compiler design books [144, 78, 145].

Alias analysis: Alias analysis is a technique in compiler theory, used to determine whether a storage location may be accessed in more than one way. In C language, two pointers are aliased if they point to the same location. For example, in this code snippet, the compiler after the alias analysis assumes that the `*p` and `i` point the same memory position. Thus, the `*p` and `i` alias.

```
int *p, i;
p = &i;
```

Alias analysis techniques are usually classified by flow-sensitivity and context-sensitivity. They may determine may-alias or must-alias information. The term alias analysis is often used interchangeably with term points-to analysis, a specific case.

Blocking Factor: Blocking Factor (BF) is layout qualifier that dictates the number of successive elements placed on the same UPC thread. In UPC language the blocking factor can be set with two ways: statically and dynamic [9]. The programmer sets the blocking factor by using the layout qualifier in the array declaration. The code snippet presents declaration examples of statically allocated shared arrays:

```
shared [8] int A[16];
shared [*] int B[16];
shared [1] int C[16];
shared [0] int D[16];
```

The programmer can also use the `upc_all_alloc()`, and `upc_global_alloc()` runtime calls to allocate memory with the desired blocking factor.

Cache: Cache or Processor Cache [146] is a smaller and faster memory which stores copied of frequently used data from the main memory.

Cache Miss Ratio: When the processor access a memory location that is not in the cache, it is called a cache miss. The processor then wait for the data to be fetched from either the next cache level or the main memory. To calculate the cache miss ration we can use this equation:

$$miss\_ratio = \frac{Missed\_accesses}{Total\_memory\_accesses}$$

The cache misses categorized into two groups:

- Local miss rate misses in this cache divided by the total number of memory accesses to this cache (Miss rate L2).
- Global miss rate misses in this cache divided by the total number of memory accesses generated by the CPU ($Miss\_RateL1 \times Miss\_RateL2$)

Common subexpression elimination: Common subexpression elimination is a compiler optimization that searches for instances of identical expressions, and replaces them with a single variable holding the computed value. Figure A.1 presents an example of the optimization. Note that the compiler has to calculate if the memory cost of the store to `temp` is less than the cost of the multiplication.

Programmer's Code	After common subexpression elimination

`a = b * c + d;`
`e = b * c * f;`
	`temp = b * c;`
	`a = temp + d;`
	`e = temp * f;`

Figure A.1: Example of common subexpression elimination.

Constant propagation: Constant folding and constant propagation are related compiler optimizations used by many modern compilers. An advanced form of constant propagation known as sparse conditional constant propagation can more accurately propagate constants and simultaneously remove dead code. Figure A.2 presents an example of constant propagation optimization. In this example, the constant value of 'a' is propagated through the statements simplifying the return expression.

Control Flow Graph: Control Flow Graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

Programmer's Code	After constant propagation

`int app_kernel(int c){`
`  int a = 10;`
`  int b = 9 - (a / 2);`
 	
`  if (c > 10) {`
`  c = c - 10;`
`  }`
`  return c * (60 / b) + a;`
`}`
	`int app_kernel(int c){`
	`  int b = 4;`
	
	`  if (c > 10) {`
	`  c = c - 10;`
	`  }`
	`  return c * 15 + 10;`
	`}`

Figure A.2: Example of constant propagation optimization.

Data Flow Graph: Data Flow Graph or Diagram (DFD) is a graphical representation of the data "flow" through the application code, and models its process aspects.

Data Dependencies: When two program statements (or instructions) refers to the data of a preceding statement, then there are data dependencies between them. There are four type of data dependencies:

- Input dependence (RAR) occurs when statements S_1 and S_2 both read from the same location. For example:

```
S1: A = B
S2: C = B
```

- Output dependence (WAW) occurs when two statements both write to the same location. For example statements S_1 and S_3 write on the same location:

```
S1: B = 1
S2: C = B + 24
S3: B = 7
```

- Flow dependence (RAW) occurs when S_1 writes to a location and a subsequent statement S_2 reads from the same location. For example:

```
S1: B = A
S2: C = B
```

- Anti-dependence (WAR) occurs when S_1 reads from a location and a subsequent statement S_2 writes to the same location. For example:

```
S2: A = B + 24
S2: B = 7
```

Dead Store: Dead store statements are the statements that assign a value to a variable, but this variable is not used by any subsequent statement or instruction.

Dead Stores are wasteful of processor time and memory. The compiler eliminates them through the use of static program analysis and dead store elimination optimization.

High Radix Interconnect: High Radix Interconnects are the networks with switches or routers that contain large number of skinny ports. The motivation behind this trend is that the bandwidth between nodes increases, however, the latency between nodes does not. Thus, the idea is to create router with a large number of skinny ports in order to reduce the latency between nodes.

Induction Variable: Induction variable is a scalar variable whose value gets increased or decreased by a fixed constant amount on every iteration of a loop.

Iteration count: Put simply, the iteration count of a loop is the number of times that the loop body get executed.

Loop-Invariant Code Motion: Loop-invariant code motion (also called hoisting or scalar promotion) is a compiler optimization which moves of statements or expressions outside the body of a loop without affecting the semantics of the program. Figure A presents an example of the loop-invariant code optimization. In this example, the calculation of expression `x = y + z` is independent of the induction variable of the loop. Moreover, the expression `x * x` depends on the value of x that is also independent. Thus, the compiler optimization moves these expressions before the loop body.

Programmer's Code	After optimization

`for (int i = 0; i < n; i++) {`
`  x = y + z;`
`  ARRAY[i] = i + i * z + x * x;`
`}`
	`x = y + z;`
	`t1 = x * x;`
	`for (int i = 0; i < n; i++) {`
	`  ARRAY[i] = i + i * z + t1;`
	`}`

captionExample of constant invariant movement optimization.

Loop unrolling: Loop unwinding or unrolling is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size by reducing instructions that control the loop. The loop transformation can be undertaken manually by the programmer or by an optimizing compiler.

Network Contention: This term describes the situation when two or more entities are competing for shared resources. The term is used especially in networks to describe the situation where two or more nodes attempt to use the limited bandwidth of a channel at the same time.

Normalized Loop:A loop which the loop variable starts from 0 and gets incremented by a constant value at every iteration until the exit condition.

Static Single Assignment (SSA) form: SSA form an intermediate representation, where each variable is assigned exactly once. Variables from the initial program are split into versions. The new variables are typically renamed according to the original name accompanied a subscript. SSA form is normally used as an extension on the top of the compiler's intermediate representation, and it is used in various optimizations.

Strip-Mining: Loop Strip-Mining, Loop Blocking, or Loop tiling is a compiler optimization that partitions a loop's iteration space into smaller blocks or chunks, to improve the cache locality. Figure A.3 provides an example of the loop transformation. The `BLK` variable is called block size of the transformation.

Programmer's Code	After transforamtion

`for(i=0; i<N; ++i){`
`  ...`
`}`
	`for(j=0; j<N; j+=BLK){`
	`  for(i=j; i<min(N, j+B); ++i){`
	`  ....`
	`  }`
	`}`

Figure A.3: Example of loop blocking transformation.

Vectorization: Vectorization is the process of code optimization, during which the programmer or the compiler transforms a computer program that uses scalar variables into an implementation that uses one operant with multiple data. Automatic compiler vectorization is a special case of automatic parallelization, where the compiler converts part of the code to vector implementation that uses multiple data with one operant or instruction.