Computer Science Technical Report

A Performance Model for Unified Parallel C

Zhang Zhang

Michigan Technological University
Computer Science Technical Report
CS-TR-07-04
June, 2007

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Abstract

This research is a performance centric investigation of the Unified Parallel C (UPC), a parallel programming language that belong to the Partitioned Global Address Space (PGAS) language family. The objective is to develop performance modeling methodology that targets UPC but can be generalized for other PGAS languages.

The performance modeling methodology relies on platform characterization and program characterization, achieved through shared memory benchmarking and static code analysis, respectively. Models built using this methodology can predict the performance of simple UPC application kernels with relative errors below 15%. Beside performance prediction, this work provide a framework based on shared memory benchmarking and code analysis for platform evaluation and compiler/runtime optimization study. A few platforms are evaluated in terms of their fitness to UPC computing. Some optimization techniques, such as remote reference caching, is studied using this framework.

A UPC implementation, MuPC, is developed along with the performance study. MuPC consists of a UPC-to-C translator built upon a modified version of the EDG C/C++ front end and a runtime system built upon MPI and POSIX threads. MuPC performance features include a runtime software cache for remote accesses and low latency access to shared memory with affinity to the issuing thread. In this research, MuPC serves as a platform that facilitates the development, testing, and validation of performance microbenchmarks and optimization techniques.

Contents

Chapter 1

Introduction

High performance computing has long been dependent on the promised high performance of multiprocessor systems. There are a number of problems, however, making the potential of multiprocessor systems difficult to realize. One of the primary problems is that programming these systems is very difficult because of the lack of good parallel programming models.

During the quest for good parallel programming models, there have emerged two major categories of models that map onto two major categories of parallel architectures. The shared memory programming model is a reflection of shared memory architectures. It offers implicit parallel expression and a single address space for a collection of processes running in parallel. Algorithm design and analysis in this model is simple because it resembles a sequential programming model. But the shared memory programming model is rarely applied to large scale parallel computation tasks because shared memory architectures suffer from serious scalability issues when the number of processors grows. On the other hand, the message passing programming model reflects distributed memory architectures, which are scalable to very large numbers of processors and are more widely used. The problem is programming with this model requires explicit expression of communications, making parallel programming very tedious and program analysis much obscured.

When the distributed memory architectures gained popularity since they are more scalable and affordable, so did the message passing programming model. For example, MPI has become the *de facto* standard interface for large scale parallel and distributed system programming. However, the simplicity provided by the shared memory programming model is still attractive. This situation has led to many researches on new parallel programming models that exploit the advantages of explicit parallelism while still providing a single address space. The emerging Partitioned Global Address Space (PGAS) languages are the most recent response to the problem. The three prominent members of the current PGAS languages family are Unified Parallel C [CDC+99], Co-Array Fortran [NR98], and Titanium [H*et al.*01]. They are developed by seperate research groups as language extensions to ANSI C, Fortran and Java, respectively.

PGAS languages realizes a programming model with both shared memory and distributed memory flavors. They are based on a conceptual partitioned shared memory model. At the center of this model is a global address space partitioned among a collection of processes. Parallel execution streams are mapped onto multiple processes. Each process is associated with a disjoint block of memory. Each block has a *shared* part that is exposed and visible by all processes and a *private* part that is accessible by the owner process only. Processes reference any positions in the *shared* space using a common syntax. References made within a processes's own partition have the least cost. This model retains the simplicity of the shared memory programming model but gives programmers the control over data locality and memory consistency. Besides, PGAS languages follows the following philosophy:

1. They support both distributed memory machines and shared memory machines. The goal is to provide performance comparable to MPI on a wide range of parallel platforms.

2. They encourage programmers to be aware of characteristics of underlying architecture, for example, memory latency and bandwidth, when designing parallel algorithms.

3. They provide concise and efficient syntax, fostering simplicity in algorithm design and analysis.

This work intends to be a performance-centric study on the PGAS programming model and implementations, using UPC as a representative of all PGAS languages.

Among all PGAS languages, Unified Parallel C (UPC) has been gaining interest from academia, industry and government labs. UPC was recognized by the high-performance computing community [Ree03] as part of an revolution to answer the challenges posed by future petascale parallel computers. A UPC consortium [Geo04] has been formed to foster and coordinate UPC development and research activities. Several UPC implementations are available, including commercial implementations. These implementations cover a wide range of parallel architectures and operating systems.

This work first describes the implementation of MuPC, a UPC runtime system based on MPI and POSIX threads. The emphasis is put on various performance improving techniques and trade-offs between performance and simplicity. Then, performance benchmarking results for MuPC are reported, alongside with the results obtained for other UPC implementations. This study designs a few synthetic micro-benchmarks that are particularly suitable for UPC, which provide more insights into UPC application performance than traditional message passing oriented benchmarks do.

The foremost goal of this work, however, is to develop techniques for performance model construction for UPC applications. This work is the first to propose a theory of UPC performance modeling, which projects UPC programs's complex performance behavior onto a two-dimensional plane. On one dimension are the architectural characteristics, on the other are the shared memory access patterns of a program. Architectural characteristics that have an impact on program performance are identified and abstracted using a few simple metrics. The work then goes on to design benchmarks to measure these metrics. Shared memory access patterns that may appear in a program are categorized into several typical patterns. Performance models are then constructed by combining the effects of architectural characteristics and the effects of shared memory access patterns.

A performance model based on this theory reveals performance loopholes in UPC compilers and run time systems, as well as loopholes inherited from applications algorithm design. By doing this, the model is able to identify techniques that compilers, run time systems and programmers can use to improve the performance. Moreover, the model can predict UPC application performance on existing and future parallel architectures. It can expose those architectural features that hinder the current performance optimization efforts, and tell us what features are desired for UPC code to run efficiently.

Note that although the performance modeling in this research targets UPC, the performance model construction techniques used here can be extended to other PGAS languages due to the programming model similarity among PGAS languages and also because these languages use the same shared memory abstraction, namely the partitioned shared memory.

The rest of this report is organized as follows: Chapter 2 is an overview of the UPC language. Chapter 3 discusses existing UPC implementations and related performance benchmarking work. Chapter 4 describes the MuPC runtime system. Chapter 5 is a detailed study of the performance implications of the remote reference caching implemented in MuPC. Chapter 6 discribes a theory

of performance modeling and benchmarking. Chapter 7 validates performance models using some simple application kernels. Last, Chapter 8 summarizes findings and proposes future work.

Chapter 2

UPC background

UPC is a parallel extension of the C programming language intended for multiprocessors with a common global address space [CDC+99]. UPC provides a common syntax and semantics for explicitly parallel programming in C. It also provides efficient and easy mapping from UPC language features to the underlying architectures.

2.1 History of UPC

UPC is a collective achievement of several previous similar attempts. It has three predecessors.

- **Split-C** Split-C [CDG+93] was a small parallel extension to C targeted toward massively parallel machines with distributed memory, such as the Thinking Machine CM-5 and Cray T3D. It provides a global address space with clear syntax distinction between local and global accesses. It also provides "split-phase transaction" semantics for remote accesses to allow efficient overlapping of communication and computation. This feature evolved into the split-phase barriers in UPC. Split-C introduced two distinct synchronization semantics at global level and at processor level. Global level synchronization is a global barrier, while processor level synchronization ensures completion of store transactions local to a processor. These are the predecessors of UPC's `upc_barrier` and `upc_fence` respectively.

- **AC** AC [CD95a] is a language inspired by the same objectives that inspired Split-C. It is very similar to Split-C in terms of global address space, distinction between local and global accesses, and synchronization semantics. Moreover, it enriches the semantics of global accesses by the introduction of a "dist" type-qualifier for explicit control over array distribution. AC strictly follows the traditions of C in dealing with pointers. The combined effect of using the "dist" type-qualifier with pointers enables distribution of complex data structures. The "dist" type-qualifier later evolved into the "shared" type-qualifier in UPC, where the semantics of block size in distribution is integrated, eventually making it possible to distribute complex data structures over the global address space. AC has been implemented on Cray T3D and has achieved good performance [CD95b].

- **PCP** Parallel C Preprocessor [BGW92] is an effort to port the simple abstraction of serial computing model to shared memory and distributed memory platforms. The design of PCP is based on the belief that a viable parallel algorithm is the one that can be efficiently supported on a wide range of architectures. To achieve this, PCP adopts a so-called "Reducible Parallel

Programming Model", where operations dealing with parallel constructs on a more sophisticated architecture are reduced to simple operations on a simpler architecture. For example, memory operations on SMP's and MPP's are supported at the hardware level but they are reduced to using active messages on message passing platforms. A unique feature of PCP is the split-join execution model. Concurrent processes in a PCP program can be subdivided into teams and the teams can merge or split further. This model facilitates exploration of nested parallelism, allowing more concurrency from a fix number of processes.

In 1999, the first UPC language specification was introduced on the basis of previous work. Bill Carlson, et al [CDC+99] implemented the first specification on a Cray T3E. Several open source compilers and a commercial compiler quickly followed. At the same time, the language specification evolved. The latest UPC specification as of 2006 is UPC version 1.2 [EGCD05]. Specifications on collective operations, memory model, and parallel I/O also emerged.

The UPC development consortium [Geo04] hosted by George Washington University now has working groups from universities, national labs, and industrial, covering a wide range of UPC topics from low-level API design, performance analysis, debugging, to application and library development.

2.2 The UPC language

UPC is a superset of ANSI C (as per ISO/IEC9899 [ISO00]). This section is a summary of UPC language features.

2.2.1 Execution Model

UPC programs adopt the single program multiple data (SPMD) execution model. Each UPC *thread* is a process executing a sequence of instructions. UPC defines two constants, `THREADS` and `MYTHREAD`, to respectively denote the number of threads in a UPC program and the identity of a particular thread. `THREADS` can be a compile time constant or a run time constant. `THREADS` and `MYTHREAD` do not change during execution.

2.2.2 Partitioned Shared Memory

Data objects in a UPC program can be either *private* or *shared*. A UPC thread has exclusive access to the private objects that reside in its private memory. A thread also has access to all of the shared objects in the shared memory space. UPC provides a partitioned view for the global address space by introducing the concept of *affinity*. The whole global address space is equally partitioned among all threads. The block of global address space given to a thread is said to have *affinity* to the thread. The concept of *affinity* captures the reality that on most modern parallel architectures the latencies of accessing different shared objects are different. It is assumed that an access to a shared object that has affinity to the thread that performs the access is faster than an access to a shared object to which the thread does not have affinity. Figure 2.1 illustrates the partitioned memory model of UPC.

UPC uses the `shared` type qualifier in a declaration to describe a shared data object. Objects declared in traditional C style (such as `int x;` in the figure) are private data objects. Shared and private objects are referenced using the same syntax. Shared objects are declared at file scope. UPC uses a layout type qualifier to specify the distribution of a shared array across shared memory. The

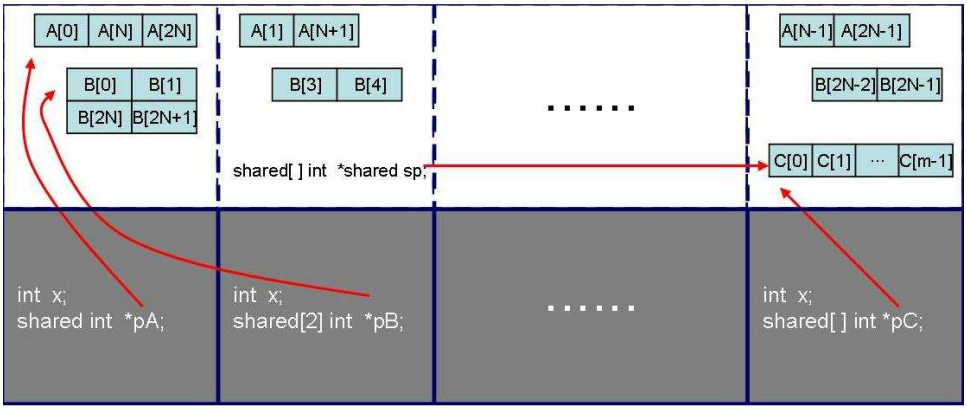

Figure 2.1: Partitioned memory model

layout type qualifier specifies a *block size*. Block size is considered a part of the data type, that is, two shared arrays with different block sizes are not type-compatible with each other. A shared array is decomposed into blocks of the given size and distributed across the blocks of the shared memory in a round-robin fashion. For example, three shared arrays are depicted in Figure 2.1. The first is declared as `shared [1] int A[2*N];`. The second is declared as `shared [2] int B[2*N+2]`. The third can only be dynamically allocated. It has the type of `shared [ ] int*`, meaning its elements all have the same affinity and its block size is indefinite.

UPC provides three types of pointers, two for pointers to shared objects and one for pointers to private objects.

- A *pointer-to-shared* is a pointer whose target is a shared data object. The pointer itself resides in the private memory space of some thread.
- A *shared pointer-to-shared* is a pointer whose target is a shared data object. The pointer itself also resides in the shared memory space.
- A *private pointer* is a conventional C pointer.

Just as in C, a pointer assumes the type of the object it points to. In UPC, a pointer-to-shared and a shared pointer-to-shared assume the block sizes of their targets. These two pointer types are illustrated in Figure 2.1. Pointer `pA`, `pB` and `pC` are pointers-to-shared. Pointer `sp` is a shared pointer-to-shared. The block sizes in their declaration govern the arithmetic of these pointers.

2.2.3 Memory consistency model

UPC provides *strict* and *relaxed* memory consistency modes. This choice matters for shared object references. The *strict* model allows very limited shared access reordering. By contrast, the *relaxed* model allows reordering and coalescence of shared object accesses as long as data dependencies within each thread are preserved. The *relaxed* model offers more opportunities for compiler and run time system optimizations.

The choice of memory consistency model can be made at three different levels.

- At the *program level*, including either the `upc_strict.h` or the `upc_relaxed.h` header file sets the default consistency model for the whole program.

- At the *statement level*, pragmas `#pragma upc strict` and `#pragma upc relaxed` change the model for shared references in a statement block, for a translation unit, or until the next pragma.
- At the object level, declaring a shared object using the keyword `strict` or `relaxed` stipulates the model for all references made to the object.

2.2.4 Other features

UPC provides a parallel loop, `upc_forall`, to facilitate job sharing across threads. `upc_forall` distributes among threads the iterations of a loop operating on a shared object based on the affinity expression of the loop.

UPC provides two types of global barriers. `upc_barrier` is similar to the `MPI_Barrier` operation in MPI. `upc_notify` and `upc_wait` together offer the semantics of split-phase synchronization. This feature is inherited from the split-phase transactions in Split-C.

UPC also provides thread-level synchronization through `upc_fence`. All accesses to shared objects made by a thread before the fence must be completed before any accesses to shared objects after the fence can begin. The fence only affects the actions of the calling thread.

Other features include locks (to synchronize accesses to shared objects by multiple threads). A set of collective routines [WGS03], dynamic memory management, and "string" functions are also available in UPC. Table 2.1 comprehensively summarizes the UPC language API.

Type qualifiers	`shared, shared[ ]`
Thread identifiers	`THREADS` `MYTHREAD`

Shared data operators	`upc_localsizeof()`
	`upc_blocksizeof()`
	`upc_elemsizeof()`
	`upc_threadof(x)`
	`upc_phaseof()`
	`upc_addrfield()`
	`upc_resetphase()`
	`upc_affinitysize()`

Job sharing	`upc_forall()`
Global synchronization	`upc_barrier()`
Split phase synchronization	`upc_notify()` `upc_wait()`
Thread-level synchronization	`upc_fence()`

Locks	`upc_global_lock_alloc()`
	`upc_all_lock_alloc()`
	`upc_lock_free()`
	`upc_lock()`
	`upc_lock_attempt()`
	`upc_unlock()`

Dynamic memory management	`upc_global_alloc()`
	`upc_all_alloc()`
	`upc_alloc()`
	`upc_free()`

Memory manipulation	`upc_memcpy()`
	`upc_memget()`
	`upc_memput()`
	`upc_memset()`

Collectives	`upc_all_broadcast()`
	`upc_all_scatter()`
	`upc_all_gather()`
	`upc_all_gather_all()`
	`upc_all_exchange()`
	`upc_all_permute()`
	`upc_all_reduce()`
	`upc_all_prefix_reduce()`

Table 2.1: UPC language API

Chapter 3

Related work

UPC is a maturing language. Researchers have mostly been focusing on language development and extensions design [CDC+99, WGS03, KW04, EGCS+03, Sav02, BBC+03, BB03, UC 04b, BD03, CKY03]. This section introduces existing UPC compilers and implementations, and related performance studies.

3.1 UPC implementations

A variety of open source and commercial implementations have been available to support UPC on shared memory architectures, SMP clusters, and Beowulf-type Linux clusters.

3.1.1 Cray UPC

The first UPC was developed by Bill Carlson, *et al.* [CDC+99] on a Cray T3E. It is a partial implementation of UPC Specifications V1.0. It lacks dynamic allocation routines and string handling routines. No further evolution of this implementation is expected. Instead, a full native implementation of UPC has been integrated into the Cray X1 system [Cra03].

3.1.2 Berkeley UPC

The most widely used public domain UPC compiler is Berkeley UPC [UC 04a]. It has a highly portable runtime system because it provides a multi-layered system design that interposes the GASNet communication layer between the runtime system and the network hardware [UC 04b]. Core and extended GASNet APIs can be mated with a variety of runtime systems, such as UPC and Titanium, on one side and with various types of network hardware, such as Myrinet, Quadrics, and even MPI, on the other side. Berkeley UPC includes a UPC-to-C translator based on the Open64 open source compiler. Various optimizations, mainly for generating shared memory latency tolerant code, are done at the UPC source code level. Translator-level optimizations for Berkeley UPC are described in [CBD+03]. Contrary to the runtime system, the translator is not portable. As a result, users of Berkeley UPC typically have to send their code to a translator proxy (an HTTP service) set up by the Berkeley group for compilation, then get back the translated C code, which is then compiled and run on local machine.

3.1.3 Intrepid UPC

Intrepid Technology provides a UPC compiler [Int04] as an extension to the GNU GCC compiler (GCC 3.3.2). It supports only shared memory platforms and SMP platforms such as the SGI Irix, Cray T3E, and Intel x86 uniprocessor and SMPs. It is freely available under the GPL license.

3.1.4 HP UPC

Hewlett-Packard offered the first commercially available UPC compiler [Hew04]. The current version of this compiler targets Tru64 UNIX, HP-UX, and XC Linux clusters. Its front end is also based on the EDG source-to-source translator. The Tru64 runtime system uses the Quadrics network for off-node remote references. Runtime optimizations include a write-through runtime cache and a trainable prefetcher.

3.1.5 MuPC

MuPC [Sav02] is a run time system developed by Michigan Technological University with help from the HP UPC group. MuPC implements an API defined by a UPC-to-C translator contributed by HP. The translator is based on the EDG C/C++ front end. The run time system is implemented using MPI and the POSIX standard threads library. MuPC currently supports AlphaServer SMP clusters and Intel X86 Linux clusters. MuPC is the subject of the next chapter of this report.

3.2 Performance studies

Performance studies done so far fall into two categories: benchmarking and run time system (or compiler) monitoring.

3.2.1 Performance benchmarking

El-Ghazawi, *et al.* [EGC01] developed the first set of performance benchmarks for UPC. This set consists of a synthetic benchmark suite and an application benchmark suite. The synthetic benchmark characterizes the performance of memory accesses, including private accesses, local-shared accesses and remote-shared accesses. The application benchmark contains three application tests representing three typical parallel processing problems: (1) Sobel edge detection, an image processing problem, (2) matrix multiplication, a dense matrix algorithm, and (3) the NQueens problem, an embarrassingly parallel algorithm. *UPC_Bench* helped expose the first performance loophole recognized by UPC compiler developers. The loophole exists in the early releases of HP UPC compiler for AlphaServer platforms, where the local-shared memory references are implemented to be almost as expensive as the remote-shared memory references. *UPC_Bench* also investigated remote reference prefetching and aggregating, two potential compiler optimization techniques.

Cantonnet and El-Ghazawi [CEG02] studied the performance of UPC using the NAS Parallel Benchmark (NPB) [BBB94]. The five kernels (CG, EP, FT, IS and MG) in the NPB suite were rewritten in UPC. Their performance was measured and compared with that of the MPI implementation on an AlphaServer SC machine running HP UPC. Results showed that UPC has a potential to perform as well as traditional parallel computing paradigms such as MPI, only if the compilers are able to optimize local-shared accesses and to aggregate fine-grained remote-shared accesses.

3.2.2 Performance monitoring

Performance of various UPC compilers and runtime systems have been closely monitored. El-Ghazawi, *et al.* [EGC01, CEG02] investigated the performance of the HP UPC compiler on an AlphaServer SC platform. Cantonnet, *et al.* [CYA+04] evaluated the GCC UPC implementation on SGI Origin family NUMA architectures. Chen, *et al.* [CBD+03] studied the performance of the Berkeley UPC compiler on the an AlphaServer SC.

Zhang and Seidel [ZS05] did performance analysis for UPC on Linux clusters where the only two UPC implementations supporting Linux platforms, MuPC and Berkeley UPC, were evaluated. This work also evaluated MuPC on an AlphaServer SC and compared the performance with Berkeley UPC and HP UPC.

Besides performance monitoring and benchmarking, Prins, *et al.* looked at performance issues from the perspective of language features. Their work [BHJ+03] analyzed the UPC language features that thwart the performance on current parallel architectures, especially on clusters. They held a pessimistic view about the fine-grain communication pattern inherent to UPC. However, they predicted that a *hybrid* programming model mixing coarse-grain and fine-grain communication may improve the performance of UPC and still retain its ease-of-use feature. Prins, *et al.* also pioneered the search for a killer application for UPC. Their work [PHP+03] on the implementation of unbalanced tree search showed that the UPC implementation is shorter in length than the MPI implementation, and the UPC implementation performs better than the MPI implementation on a SGI Origin 2000 platform.

Other performance related work includes an attempt to improve performance by overlapping communication and computation [IHC04], and a study of exploiting UPC's relaxed memory consistency model [KW04]. In the first work, Iancu, *et al.* discussed how to segment large remote references so that transfers of small-sized segments can be pipelined and overlapped with local computation to save communication cost. They used the technique of *message strip mining* combined with *loop unrolling* to optimize vectorizable parallel loops. In the second work, Kuchera and Wallace contemplated how a few loop optimization techniques (loop fusion, loop fission, and loop interchange) commonly adopted in sequential programming language compilers could also improve UPC performance by taking advantages of a less constrained memory consistency model.

Chapter 4

MuPC: A UPC runtime system

A substantial part of this research is the development of a UPC implementation called MuPC [Mic05]. This work used MuPC as a platform to develop, test, and validate all microbencmarks that are used in performance study described in later chapters. MuPC also served as a sandbox for developing runtime optimization techniques, such as remote reference caching.

MuPC contains a runtime system and a UPC-to-C translator. The translator was developed with help from the High Performance Technical Computing Division at Hewlett-Packard Company. The runtime system is based on MPI and the POSIX standard threads library. It implements an API defined by the UPC-to-C translator.

Each UPC thread at run time is mapped to two Pthreads, the *computation* Pthread and the *communication* Pthread. The translator is a modified version of the EDG C/C++ V3.4 front end [Edi05]. This report mainly focuses on the internal structure of the runtime system. The work related to the EDG front end modification is also discussed.

The MuPC runtime system (excluding the EDG front end) is portable to any platform that supports MPI and POSIX threads. This means that a wide range of architectures and operating systems can run MuPC. MuPC has been ported to Intel x86/Linux, HP AlphaServer/Tru64 Unix, and Sun Enterprise (Solaris). The only portability issue arises from the UPC-to-C translator, which is released as a binary executable. Executables for platforms other than the aforementioned three are available upon request.

4.1 The runtime system API

The MuPC runtime system is a static library that implements a UPC runtime system API. The front end translates a UPC program into an ANSI C program where UPC-specific features are represented using structures and function calls defined in the API. For example, pointers-to-shared are translated into `_UPCRTS_SHARED_POINTER_TYPE` structures. Remote memory reads and writes are translated into calls to `_UPCRTS_Get()` and `_UPCRTS_Put()`, respectively. The translated code is compiled using a C compiler such as `icc`, then linked with the runtime system library to generate an executable. The executable is then run as an MPI program. This section describes the runtime system API and discusses some high-level decisions made in its design.

The runtime system API used by MuPC is based on an API originally defined by the UPC development group at Hewlett-Packard. Despite the similarity between the two APIs, their implementations are completely different. While MuPC implements the API using MPI and Pthreads, HP's implementation is based on the NUMA architecture of the SC-series platform and uses the Elan

communication library and Quadrics switch to move data between nodes. Compared with MPI, the Elan library provides much lower level message passing primitives and higher performance. On the other hand, the MPI+Pthreads implementation gives MuPC greater portability. MPI also makes the implementation simpler because it already guarantees ordered message delivery in point-to-point communication. In addition, process startup, termination and buffer management are straightforward in MPI.

4.1.1 Runtime constants

`THREADS` and `MYTHREAD` are runtime constants representing the number of UPC threads and the ID of a particular running thread, respectively. They are available in the runtime library through two global constants: `_UPCRTS_gl_cur_nvp` and `_UPCRTS_gl_cur_vpid`.

4.1.2 Shared pointer representation

`_UPCRTS_SHARED_POINTER_TYPE` is a C structure in the API that represent pointers-to-shared. A pointer-to-shared contains three fields: *address*, *affinity* and *phase*. The *address* and *affinity* fields together specify the location of an object in shared memory. The *phase* field specifies the relative offset of an element in a shared array. The *phase* field is useful only in pointer arithmetic, which is handled in the UPC-to-C translation step. In other words, translated code does not include any pointer arithmetic. The runtime system needs only *address* and *affinity* to retrieve and store shared data.

4.1.3 Shared memory accesses

Shared memory `read` and `write` operations for scalar data types are implemented using corresponding `get` and `put` functions defined in the runtime API. One key design feature of the runtime system is that all `get` operations are synchronous, while `put` operations are asynchronous. A `get` routine initiates a request to get data from a remote memory location. Each request is then completed by one of the `getsync` routines (depending on the data type), which guarantees the data are ready to use once it returns. In contrast, there are no routines to complete `put` operations. A thread considers a `put` operation to be complete as soon as the function call returns, although the data to be written are likely still in transit to their destination. This permits the MuPC runtime system to issue multiple `put` operations at a time, but only one `get` at a time. It is desirable to have the runtime issue multiple `gets` at a time and complete them altogether with just one `getsync` routine, but this requires a dependence analysis within a block to determine which `gets` can be issued simultaneously. Basically, only a sequence of `gets` without the WAR hazards in between can be issued simultaneously. This feature is not currently implemented in MuPC.

Another set of corresponding runtime functions are defined for moving non-scalar data objects between shared memory locations. Non-scalar types, such as user defined data types, are treated by the runtime system as raw bytes. Both `block get` and `block put` operations are synchronous. A `block get` or `block put` request is issued first, then it is completed by a matching completion routine. In the `block get` case, the completion routine guarantees the data retrieved are ready to be consumed by the calling thread. In the `block put` case, the completion routine guarantees the source buffer where the data came from is ready to be overwritten because the data are already on the way to their destination.

4.1.4 One-sided message passing routines

Besides shared memory accesses, UPC provides one-sided message passing functions: `upc_memcpy()`, `upc_memget()`, `upc_memput()` and `upc_memset()`. These functions retrieve or store a block of raw bytes from or to shared memory. The MuPC runtime supports these directly by providing corresponding runtime functions. These functions rely on the aforementioned `block get` and `block put` functions to accomplish data movement.

4.1.5 Synchronization

The MuPC runtime system directly supports the UPC synchronization functions by providing matching runtime routines. It is noteworthy that a UPC barrier is not implemented using `MPI_Barrier()`, although it appears to be natural to do so. UPC barriers come with two variations, *anonymous* barriers and *indexed* barriers. An *indexed* barrier includes an integer argument and can be matched only by barriers on other threads with the same argument or by anonymous barriers. MPI barriers are incapable of supporting complex synchronization semantics as such, so the MuPC runtime system implements its own barriers, using a binary tree-based algorithm.

A `fence` in UPC is a mechanism to force completion of shared accesses. In the MuPC runtime system, `fences` play a significant role in the implementation of UPC's memory consistency model. More details are discussed in section 4.4.7.

4.1.6 Locks

UPC provides locks to synchronize concurrent accesses to shared memory. Locks are shared opaque objects that can only be manipulated through pointers. In MuPC, the lock type is implemented using a shared array with size `THREADS` and block size 1. Each lock manipulation routine, such as `upc_lock`, `upc_unlock`, and `upc_lock_attempt`, has a corresponding runtime function.

4.1.7 Shared memory management

Static shared variables are directly supported using static variables of ANSI C. For static shared arrays distributed across threads, the translator calculates the size of the portion on each thread, and accordingly allocates static arrays on each thread. For static shared scalars and static shared arrays with indefinite block size, the translator replicates them on all UPC threads but only the copy on thread 0 is used. This ensures that on each thread the corresponding elements of a shared array that occupy multiple threads always have the same *local* addresses. This is a desirable feature that greatly simplifies the implementation of remote references.

The memory module of the runtime system manages the shared heap. At start-up, the memory module reserves a segment of the heap on each thread to be used in future memory allocations. The size of this segment is a constant that can be set at run time. The starting address of this reserved segment is the same on all threads. This guarantees that corresponding elements of an allocated array have the same local addresses on each thread. The memory module uses a simple *first-fit* algorithm [Tan01] to keep track of allocated and free space on the reserved segment on each thread at run time.

4.1.8 Interface extension

One goal of MuPC is to provide an experimental platform for language developers to try out new ideas. Thus the runtime API is extensible. One extension that is currently being investigated is the implementation of atomic shared memory operations, which are not part of the latest UPC language specifications. Section 4.4.10 contains more details about implementing atomic operations.

4.2 The communication layer

Bonachea and Duell [BD03] presented five requirements for communication layers underlying the implementation of global address space languages such as UPC:

1. Support remote memory access (RMA) operations (one-sided communication).
2. Provide low latency for small remote accesses.
3. Support nonblocking communication.
4. Support concurrent and arbitrary accesses to remote memory.
5. Provide or support the implementation of collective communication and synchronization primitives.

MPI was chosen to implement the MuPC runtime API for two important reasons: (1) The prevalence of MPI makes the runtime system easily portable. (2) MPI libraries provide high-level communication primitives that enable fast system development by hiding the details of message passing. MuPC is built on the top of only a handful of the most commonly used MPI routines, such as `MPI_Isend()` and `MPI_Irecv()`.

However, MPI does not meet the five requirements listed above. Specifically, MPI-1.1 supports only two-sided communication. The one-sided get/put routines defined in the MuPC runtime API must be simulated using message polling because the receiver does not always know when a sender will send a message. This justifies a two-threaded design as discussed in the next section. The MPI-2 standard supports one-sided communication with some restrictions, but the restrictions make the one-sided message passing routines incompatible with the requirements of UPC [BD03]. In addition, MPI-2 has not been as widely implemented as MPI-1.1. Therefore, MuPC is currently implemented with MPI-1.1.

The MuPC runtime system constructs its own MPI communicator. All UPC threads are members of the communicator `MUPC_COMM_WORLD`. MuPC uses only asynchronous message-passing functions to achieve message overlapping and to minimize message overhead.

As a user-level communication library, MPI has a higher overhead comparing to lower-level libraries such as GM (Myrinet), elan (Quadrics) and GasNet. But performance study shows that the communication layer is not the bottleneck of MuPC performance. In other words, choosing a lower-level communication library would not have led to better performance for MuPC.

4.3 The two-threaded implementation

MuPC uses a two-threaded design to simulate one-sided communication, or remote memory access (RMA), using MPI's two-sided communication primitives. This design uses POSIX Threads

(Pthreads) because of its wide availability. Each UPC thread spawns two Pthreads at runtime, the *computation* Pthread and the *communication* Pthread. The computation Pthread executes the compiled user code. This code contains calls to MuPC runtime functions. The computation Pthread delegates communication tasks to the communication Pthread. The communication Pthread plays two basic roles. First, it handles the read and write requests of the current UPC thread by posting `MPI_Irecv()` and `MPI_Isend()` calls and completing them using `MPI_Testall()`. Second, it responds to the read and write requests from remote threads by sending the requested data using `MPI_Isend()` or storing the arriving data directly into memory. Since the communication Pthread has to respond to remote read and write requests in real time, it implements a polling loop (by posting persistent MPI receive requests) that listens to all other UPC threads. Figure 4.1 depicts the operation of the communication Pthread. The two Pthreads synchronize with each other using a

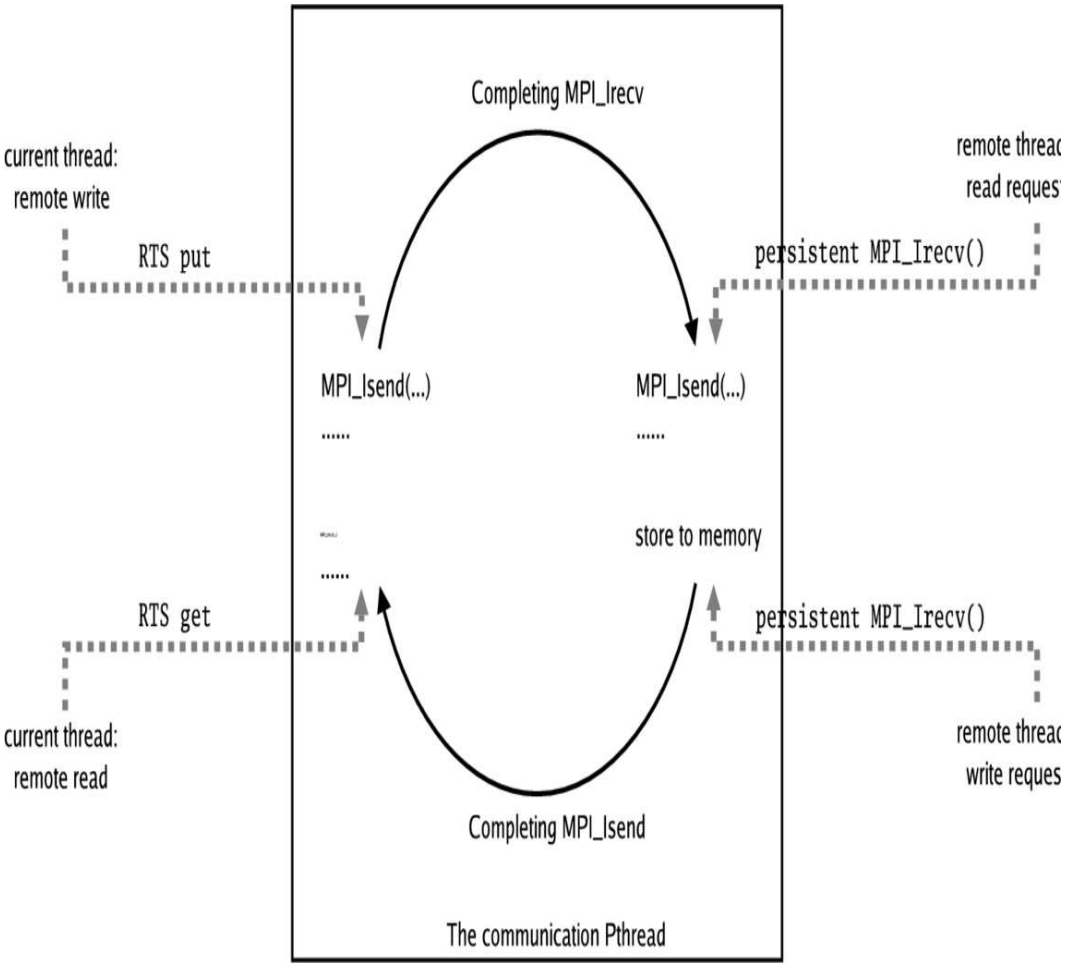

Figure 4.1: The communication Pthread

set of mutexes and conditional variables; they communicate with each other using a set of global buffers and queues. The level of thread-safety provided by MPI implementations is highly variable, so the MuPC runtime system encapsulates all MPI function calls in the communication Pthread.

4.4 MuPC internals

4.4.1 Runtime source code structure

The runtime system kernel is organized into 3 modules. The main module (rts.c) implements all runtime API functions, as well as the communication Pthread and the computation Pthread. A memory module (mem.c) defines allocation and de-allocation routines that underlie UPC's dynamic shared memory allocation functions. A cache module (cache.c) performs remote reference caching to help reduce the long latencies of referencing remote scalar shared variables.

The source code is compiled into a static library, `libmupc.a`. User code written in UPC is first translated into intermediate code by the EDG front end. Then the intermediate code is compiled using `mpicc` and linked to the runtime library by passing the option `-lmupc`.

The source package contains wrappers that encapsulate the procedures of translating, compiling, linking and running user code. The package is built using the GNU build system. Configuration scripts and makefiles are automatically generated by tools including `autoconf` and `automake`.

4.4.2 Program start and termination

The runtime system goes through the following steps to start the execution environment:

1. MuPC relies on the MPI command `mpirun` to boot the execution environment. All MPI processes spawned by `mpirun` make up a unique communicator. Each MPI process corresponds to a MuPC runtime instance. The size of the communicator is equal to the constant `THREADS`.
2. A runtime instance starts as a normal MPI process by calling `MPI_init()`. The process reads the runtime configuration file to set appropriate runtime constants. For example, the geometry of the remote reference cache.
3. Each process then initializes runtime buffers, mutexes, conditional variables, etc. These are used for communication and synchronization between the two internal Pthreads.
4. Next, remote reference cache and memory management subsystems are initialized.
5. Each process creates and starts the computation Pthread and the communication Pthread. From this point on, all MPI communication is restricted to the communication Pthread.
6. The computation Pthread starts to execute the `main()` function of the user program. A `upc_barrier` is forced by the runtime at the beginning of the user program. At the same time, the communication Pthread starts the polling loop.

The termination of a program involves the following steps:

1. When the computation Pthread exhausts the statements of the user program, it forces a `upc_barrier`, then signals the communication Pthread about job completion. After this, the computation Pthread exits.
2. On receiving the signal, the communication Pthread finishes handling the final barrier, stops the polling loop, cancels and de-allocates all persistent MPI requests, then exits.
3. The runtime's main() process waits for the termination of the two Pthreads, then de-allocates all runtime buffers, mutexes, conditional variables, the remote reference cache, and the memory management subsystem.
4. The main process calls `MPI_Finalize()` to terminate the MPI process.

4.4.3 MPI messaging

Since communication is handled by MPI message passing routines, the runtime system defines a uniform message format for all messages, including both data transfer messages and control messages. Messages for different purposes are distinguished by message types. The runtime system defines approximately 40 message types.

A message is a sequence of raw bytes and it is transfered as the data type `MPI_BYTE`. Along with the data item to be transfered, a message consists of six other fields: *message type*, *source*, *destination*, *context tag*, *data size*, and *data address*. Not all fields are used all the time. For example, control messages such as synchronization messages usually use only the first three fields (type, source and destination). Except for the data item field, all fields have fixed lengths. A message is always made 8-byte aligned to obtain optimal buffering overhead [SG98].

The runtime system maintains two global queues for message passing, one is for sending fixed size data and control messages, and the other is for sending block data messages with variable size. The difference between these two queues is that the first queue is statically declared with fixed size, while the second one is dynamically allocated at run time. Coupled with the computation Pthread and the communication Pthread, these queues provide a producer-consumer mechanism. The computation Pthread produces messages to be added to these queues, and the communication Pthread consumes the messages by posting nonblocking asynchronous MPI sends to send them. Both queues can hold a finite number of messages. Instead of being consumed one by one, they are processed in batches. The communication Pthread sends all queued messages at one time, using multiple nonblocking asynchronous MPI sends. It completes them (at a later time) using a single call to the completion routine `MPI_Testall()`.

Accesses to the queues are synchronized by a set of Pthread mutexes and conditional variables. If a queue is full because the communication Pthread is too slow, the computation Pthread stops waiting for vacancies on the queue. The communication Pthread periodically checks the queue to process pending messages. When the queue becomes empty the computation Pthread is signaled. On the other hand, if the communication Pthread does not have enough work to do because the queues are empty, it yields its current time slice to the computation Pthreads.

4.4.4 Read and write scalar variables

One of the most salient features of UPC programming language is that it supports *implicit* communication. Read and write accesses to shared scalar variables are expressed as variable assignments, not syntactically distinguishable from local reads and writes. The EDG translator translates remote references into calls to runtime get/put functions.

The runtime function `_UPCRTS_GetBytes()` performs shared read operations. If the shared data to be read has affinity to the local thread, the data are directly read from the local memory. Otherwise, this function builds a request message to be sent by the communication Pthread, then it calls `_UPCRTS_GetSync<Type>()` to complete the read request. This requires checking the global buffer where the communication Pthread stores the retrieved data; grabbing the data if it is available, or blocking if the data are still not ready. When the requested data arrives, the communication Pthread stores them to a global buffer and signals the computation Pthread.

On the other end, the remote thread which serves the read request hears the request message because its communication Pthread is a passive listener. Its communication Pthread fetches the requested data from local memory, packages it into an MPI message and sends it back to the requesting thread. This process involves the communication Pthread only, the computation Pthread on the remote thread is not involved.

Write operations to shared variables are performed by the runtime function _UPCRTS_Put<Type>(), where the <Type> corresponds to the type of the variable to be written to. The computation Pthread builds a message with the data being piggybacked. It appends the message to the "send" queue to be processed by the communication Pthread. The communication Pthread sends the message using MPI_Isend() and completes the send (possibly together with other sends) with a call to MPI_Testall().

The communication Pthread on the remote thread which is the destination of the write receives the message, decodes it and stores the data to local memory. Again, the computation Pthread on the remote thread is not involved.

Note that there are no _UPCRTS_PutSync<Type>() functions to complete writes. This is how write operations are different from read operations. This implies that a user program should always use a barrier to guarantee the data are stored to the destination location. Section 4.4.7 discusses more about the implementation of UPC's memory consistency model.

Discussions above do not consider the presence of the remote reference cache. If the cache is active, then the _UPCRTS_GetBytes() and the _UPCRTS_Put<Type>() functions will try to read from or write to the cache first. Message passing is necessary only in the case of a cache miss. Section 4.4.9 contains details about remote reference caching.

4.4.5 Non-scalar type references and one-sided communication

Reads and writes for non-scalar types, for example, user-defined structures, are performed by _UPCRTS_GetBlock() and _UPCRTS_PutBlock() runtime functions. These two functions also support UPC's one-sided communication functions such as upc_memget(), upc_memput(), and upc_memcpy().

More MPI messages are exchanged for implementing non-scalar type references than for scalar type references. When the computation Pthread calls get or put, it first builds a control message specifying the the type of the operation, memory addresses involved, and data size, to be sent to the remote UPC thread by the communication Pthread. After processing the control message, the communication Pthread posts an MPI_Irecv()/MPI_Isend() to actually receive or send the data. The communication Pthread on the remote UPC thread receives the control message first, then accordingly posts an MPI_Isend() or MPI_Irecv() to match the MPI_Irecv() or MPI_Isend() on the other end.

There are two more complexities. First, data of very large size are fragmented to be processed by multiple consecutive calls to the get or put functions. The threshold of the fragmentation is adjustable at installation time and depends on the particular MPI library used. It should be tuned carefully on individual installations to achieve good performance.

Second, unlike scalar get or put requests, the data to be transfered by a non-scalar get or put request are not stored in a queue. Only a pointer to the data is queued. At a put operation, this means that the communication thread copies the data directly from the user program's memory instead of from the runtime maintained queue. Therefore, a _UPCRTS_PutBlockSync() is always needed to complete a put operation, to make sure that the source location will not be modified until the data are totally copied out. A _UPCRTS_GetBlockSync() is also needed to complete a get because get operations must be synchronous.

4.4.6 Optimization for local shared references

Local shared references are accesses to shared variables that have affinity to the accessing thread. There is no communication overhead in this type of accesses, but the latency still tends to be longer

than the latency of accessing private variables [CEG02, EGC01, CYA+04, CBD+03]. The extra cost comes from the effort of handling UPC's `shared` keyword. For example, initially the UPC-to-C translator mechanically translates shared references into calls to corresponding runtime get/put functions. The overhead of the function calls accounts for the extra cost.

An optimization has been implemented in the UPC-to-C translator to distinguish local and remote shared references. Tests are inserted for each shared reference to determine if the reference is *local* at run time by comparing the thread field of the address with `MYTHREAD`. Once identified, a local shared reference is converted to regular local memory access to avoid expensive runtime function calls.

4.4.7 Synchronization primitives

The UPC synchronization primitives include fence, split and non-split barriers, and locks.

Fences and the memory consistency model
`upc_fence` is a mechanism to force the local completion of shared memory references. It forces an order between the operations before it and the operations after it. In the MuPC runtime system, it is supported by the runtime function `_UPCRTS_PutAllSync()`.

MuPC relies on `fences` to implement the `strict` and the `relaxed` shared memory accesses. For each `strict` access, the translator forces a `fence` immediately after the access to ensure that a following access will not start until the current access is issued. No `fences` are forced for `relaxed` accesses.

Reordering `relaxed` accesses may lead to performance improvement. For example, starting long-latency operations early helps reduce the waiting time. However, at this stage MuPC is not capable of reordering user program statements. And because of the non-overtaking property of MPI messages [SG98], shared memory references in MuPC always complete in program order, as long as the remote reference caching is not used. Therefore, a `fence` is just a no-op in the absence of caching. When caching is used, however, a `fence` requires the cache to be invalidated and dirty cache lines to be written back. Section 4.4.9 contains more information about cache invalidation.

Barriers
UPC provides split and non-split barriers. A split barrier is a `upc_notify` and `upc_wait` pair, supported by the runtime functions `_UPCRTS_Notify()` and `_UPCRTS_Wait()`, respectively.

At a `notify`, the computation Pthread creates a notifying message to be broadcast by the communication Pthread to all other UPC threads, then continues until the matching `wait`. Other UPC threads may pick up the notifying message at any time (because their communication Pthreads are also passive listeners), and increment a counter by one. The split barrier semantics [EGCD03] determine that any two UPC threads can differ by at most one synchronization phase. (A synchronization phase is the epoch between two consecutive `notifies`.) Thus, each UPC thread maintains two counters for barrier purposes, one for the current phase, the other for the next phase. Depending on whether the notifying message is from a faster-going thread or a slower-going thread, one of the two counters is incremented when the notifying message is received.

At a `wait`, the computation Pthread sleeps until being wakened up by the communication Pthread when the counter for the current phase has reached the value of `THREADS-1`.

When named `notify` and `wait` are used, the runtime performs extra operations to verify that the barrier arguments match between each pair of `notify` and `wait`, and across all UPC threads. The runtime also verifies the pairs of `notify` and `wait` are not nested.

A non-split barrier, `upc_barrier`, is supported by the runtime function `_UPCRTS_Barrier()`. A binary tree based broadcasting algorithm is used to implement non-split barriers. When a UPC thread reaches a barrier, the computation Pthread creates a barrier message to be sent to `Thread 0` by the communication Pthread and then goes to sleep. `Thread 0` maintains a special counter and increments it for each barrier messages it receives. Once the counter reaches the value of `THREADS-1` and `Thread 0` itself also reaches the barrier, it broadcasts a signal to all other UPC threads. The communication Pthread on each UPC thread picks up the signal and wakes the sleeping computation Pthread. This completes the barrier.

A `fence` is inserted immediately before a `notify` and immediately after a `wait`. A `fence` is also inserted immediately before a `barrier`. The remote reference cache is invalidated at fences.

Locks

Locks are `opaque` shared objects that can be accessed using pointers (handles). The affinity of a lock object does not affect its semantics and is implementation dependent. In MuPC, lock objects have affinities with `Thread 0` except that if a lock is allocated using `upc_global_lock_alloc()` then it has affinity to the calling thread. The UPC thread with which a lock has affinity is called the host of the lock.

Lock manipulations are supported by three runtime functions, `_UPCRTS_lock()`, `_UPCRTS_lock_attempt()` and `_UPCRTS_unlock()`. One feature desired in lock manipulations is *fairness*. That is, an *acquire* of a lock will eventually succeed, as long as any UPC thread held it previously always *releases* it. To achieve this feature, each UPC thread maintains a cyclic queue for locks it hosts. Every lock request the host receives is queued until the host is able to honor the request. In more detail, lock manipulations are performed in the following way:

When a UPC thread acquires a lock, the computation Pthread calls `_UPCRTS_lock()` to build a request to be sent by the communication Pthread to the lock host, then the computation Pthread goes to sleep. The communication Pthread of the host receives the request and appends it to the queue. The communication Pthread periodically serves the lock request at the head of the queue by checking the status of the requested lock. If the lock is unlocked, the host's communication Pthread honors the request by marking the status as locked, dequeues the request, and sends a message to the requester. The communication Pthread of the requester receives the message, then wakes up the blocked computation Pthread. On the other hand, if the lock is locked, the host's communication Pthread either dequeues the request and sends a deny message to the requester, if the request is a *lock_attempt*; or it keeps the request on the queue and sends no message, if the request is a *lock*. This scheme ensures that an unhonored *lock* request is always on the queue and will be checked again and again by the host until it is eventually honored.

When a UPC thread releases a lock, the computation Pthread calls `_UPCRTS_unlock()` to build a request to be sent by the communication Pthread to the lock host, then continues. The communication Pthread of the host receives the request and changes the status of the lock to be unlocked.

The UPC language defines lock operations to be consistency points. That is, memory references inside a critical section are not allowed to appear outside the critical section. Thus, a `fence` is executed immediately after a lock is successfully acquired; and immediately before a lock is released. But `fences` are not enough to force consistency points. This subtlety can be explained by an example shown in Table 4.1.

The circled numbers in Table 4.1 represent the global order in which statements are executed. The `fence` implied at the beginning of statement ② ensures only the *local* completion of statement ①. The updated value of variable `X` may actually arrive the memory location of `X` much later than statement ④, which reads in a stale value of `X`. From the point view of Thread 0, this is a violation of lock semantics.

Thread 0	Thread 1	Thread 2
`upc_lock_t LL;`		`shared int X;`
	`upc_lock(&LL);`	
	`① write X;`	
`③ upc_lock(&LL);`	`② upc_unlock(&LL);`	
`④ read X;`		
`upc_unlock(&LL);`		`⑤ X is updated;`

Table 4.1: Locks and memory consistency

The MuPC runtime takes the following approach to solve this problem. Immediately after the `fence` implied by a lock release, a dummy message is sent to every other UPC thread to which remote writes have been made inside the critical section. The lock is then released when all involved UPC threads have acknowledged the dummy messages. This approach depends on the non-overtaking property of MPI messages, if the dummy message has arrived then all messages sent before it must have also arrived. In other word, all updates to remote memory locations inside the critical section have reached their destinations.

4.4.8 Shared memory management

Static shared variables

Static shared variables are directly supported using static variables of ANSI C. For static shared arrays that span several threads, the translator calculates the size of the portion on each thread and accordingly allocates static arrays on each thread. For static shared scalars and static shared arrays with indefinite block size, the translator replicates them on all UPC threads, but only the copy on thread 0 is used. This ensures that for a static shared array spanning several threads, corresponding elements on each thread always have the same address. This is a desirable feature that greatly simplifies the implementation of remote references. For example, for the array declared as: `shared int A[THREADS]`, the element `A[MYTHREAD]` on each thread has the same local address.

Dynamic shared memory management

The runtime functions `_UPCRTS_GlobalAlloc()`, `_UPCRTS_AllAlloc()` and `_UPCRTS_Alloc()` support dynamic shared memory allocation. The function `_UPCRTS_Free()` supports de-allocation. The memory module of the runtime manages the shared heap.

It is also desired that corresponding elements on each thread have the same address for dynamically allocated arrays spanning several threads. At start-up, the memory module reserves a segment of the heap on each thread to be used for future memory allocations. The size of this segment is a constant that can be set at run time. The starting address of this reserved segment is the same on all threads.

An exception is arrays allocated using `upc_alloc()`. This routine allocates arrays with indefinite block size, all elements have affinity with the calling thread. Therefore, this routine is essentially the same as a local allocation. It takes space from the heap, but not from the reserved segment.

The memory module uses a simple *first-fit* algorithm [Tan01] to keep track of allocated and free space on the reserved segment on each thread. Tracking is done on thread 0 only because the image of the reserved segment is always the same on all threads.

When `_UPCRTS_GlobalAlloc()` is called, the size of the portion on each thread is calculated. Then a control message containing this information is sent from the calling thread to thread 0. The memory module on thread 0 marks off an appropriate amount of space from the reserved segment and sends back to the calling thread a control message containing the information about the returned address.

_UPCRTS_AllAlloc() works similarly except that it is a *collective* function. A broadcast is needed at the end of it to notify all threads of the address of the newly allocated space.

When _UPCRTS_Free() is called, a control message containing the information about the address of the space to be de-allocated is sent to thread 0. The memory module on thread 0 marks the corresponding block of the heap as free, or does nothing if it has already been de-allocated.

4.4.9 Remote reference caching

The runtime implements a software caching scheme to hide the latency of remote references. Each UPC thread maintains a noncoherent, direct-mapped, write-back cache for scalar references made to remote threads. The memory space of the cache is allocated at the beginning of execution. The length of a cache line and the number of cache lines can be set at run time using a configuration file.

The cache on each thread is divided into THREADS-1 segments, each being dedicated to one remote thread. Either a read miss or a write miss triggers the load of a cache line. Subsequent references are made to the cache line until the next miss. Writes to remote locations are stored in the cache, then are actually written back later at a cache invalidation or when the cache line is replaced. This mechanism helps reduce the number of MPI messages by combining frequent small messages into infrequent larger messages.

Cache invalidation takes place at a fence, with dirty cache lines being written back to their sources. Dirty cache lines are collected and packed into raw byte packages destined to different remote threads. The packages are transferred using _UPCRTS_PutBlock(). The communication Pthread of a receiver unpacks each package and updates the corresponding memory locations with the values carried by the package.

A cache line conflict occurs if more than one memory block is mapped to one cache line. The incumbent will be replaced and written back if it is dirty. The runtime calls _UPCRTS_PutBlock() to perform the write-back of the replaced cache line.

The runtime also provides a victim cache to mitigate the penalty of conflict misses. The victim cache has the same structure as the main cache, except that it is much smaller. When a conflict occurs, the replaced line is stored in the victim cache. It is written back when it is replaced again in the victim cache. A cache hit in the victim cache brings the line back to the main cache, and a miss both in the main cache and in the victim cache triggers the load of a fresh cache line.

The *false sharing* problem is handled by associating with each cache line a bit vector to keep track of the bytes written by a thread. The bit vector is transferred together with the cache line and is used to guide the byte-wise updating of the corresponding memory locations.

Remote reference caching is a feature designed to improve the throughput of relaxed remote shared memory accesses. It is not favorable to strict accesses because every strict access implies a fence. As described above, cache invalidation takes place at each fence. So a strict access involves the additional cost of invalidating the cache. For applications with frequent strict accesses it is better off to disable caching.

4.4.10 Atomic operations

Atomic shared memory operations are a unique feature of MuPC. They are not defined in the latest UPC language specifications.

Atomic operations provide safe *lock-free* accesses to shared memory. When an atomic operation is performed to a shared variable by multiple UPC threads, only one can succeed, all other attempts

fail and have no effects. Five atomic operations are currently implemented in MuPC. They are supported by five corresponding runtime functions.

The runtime starts an atomic operation by building a request (message) to be sent to the thread to which the targeted memory location has affinity. Upon receiving this message the thread appends the request to a cyclic queue. Thus multiple concurrent accesses from different threads are sequentialized by the queue. The access that arrives first will be performed successfully. Then later accesses will only see that the value of the concerned memory location has been changed, and they all fail. A reply message is built for each access request, specifying success or failure, and is sent back to each requester.

4.5 The UPC-to-C translator

MuPC's UPC-to-C translator is based on the EDG front end, one of the most widely used compiler front ends [Edi05]. MuPC started with EDG Version 2.45, which did not support UPC. MuPC borrowed HP-UPC's modification to provide UPC support. From Version 3.2 and up, EDG provides native UPC syntax parsing (but only up to UPC 1.0 specification). MuPC has since released to EDG Version 3.2.

The process of translating UPC code into ANSI C code consists of two phases, parsing and lowering. In the parsing phase, the front end reads source files written in UPC, checks syntax and semantics errors, and produces an IL (intermediate language) tree to represent the source program. The IL tree is a high-level interpretation of the source program because it uses constructs corresponding very closely to UPC constructs. All UPC language specific features are kept in their original form, without being translated into equivalent C code. Then, in the lowering phase, the front end translates the UPC IL constructs into ANSI C constructs. All UPC language specific features are implemented using only ANSI C features. For example, a shared pointer is implemented using a C struct with fields to represent thread, address, and offset. Shared memory operations are implemented using get/put functions defined in the UPC runtime API. The lowered code can then be compiled using any native C compilers and linked with the MuPC runtime library.

The EDG front end performs only the first phase and can only parse UPC 1.0 syntax. So the EDG Version 3.2 code was modified to provide UPC 1.1 support and an IL lowering scheme to generate ANSI C code. The work consisted of three parts.

First, the UPC syntax parsing code is isolated. Modifications needed by UPC 1.1 specific syntax were applied to the isolated code. These included:

- Addition of predefined macros such as `__UPC_STATIC_THREAD` and `__UPC_DYNAMIC_THREADS`
- Addition of new UPC operators such as `upc_resetphase` and `upc_affinitysize`
- Deprecation of the `shared [] void *` type

Second, the UPC IL was extended to add new IL entries corresponding to the new additions in UPC 1.1. This guaranteed an accurate UPC 1.1 code interpretation that facilitates correct code generation. By preserving source-correspondence information in the IL tree this modification also ensured that appropriate debugging information is produced for UPC syntax errors.

Last, the UPC lowering code was rewritten with help from HP-UPC. The lowering phase was modified to conform to the new compile-time structures and functions provided by EDG Version 3.2. It was also modified to facilitate some simple UPC code instrumentation. For example, MuPC provides a few routines that can be inserted in a UPC program to record remote memory accesses and

cache behavior. The lowering phase is able to translate these routines in source code to appropriate runtime functions.

4.6 Performance characteristics

This section uses two home-grown synthetic microbenchmarks and the NAS Parallel Benchmark suite [BBB94] to characterize the performance of the MuPC runtime system. Measurements were obtained on a variety of parallel platforms and are compared with the performance of other UPC implementations.

4.6.1 Synthetic benchmarks

The synthetic microbenchmarks used are:

- **Streaming remote access** measures the transfer rates of remote memory accesses issued by a single thread, while other threads are idle. Four access patterns are measured: stride-1 reads and writes, and random reads and writes.
- **Natural ring** is similar to the streaming remote access test, except that all threads form a logical ring and read from or write to their neighbors at approximately the same time. This benchmark is similar to the all-processes-in-a-ring test implemented in the HPC Challenge Benchmark Suite [LDea05].

4.6.2 The UPC NAS benchmark suite

The UPC NAS benchmark suite used in this work was developed by the UPC group at George Washington University [CEG02]. This suite is based on the original MPI+FORTRAN/C implementation of the the NAS Parallel Benchmark suite (NPB 2.4) [BBB94]. There are five kernel benchmarks in the NPB suite. Every benchmark comes with a "naïve" implementation and one or more "optimized" implementations. The naïve implementation makes no effort to incorporate any hand-tuning techniques. The optimized implementations incorporate to various extents hand-tuning techniques such as prefetching and privatized pointers-to-shared [CEG02]. This work measured the performance of the naïve version and the most optimized version. The class A workload was used as the input size for all benchmarks. The five benchmarks are conjugate gradient (CG), embarrassingly parallel (EP), Fourier transform (FT), integer sort (IS), and multigrid (MG).

4.6.3 Results

Measurements were obtained on two platforms, an HP AlphaServer SC SMP cluster and a Linux cluster connected with a Myrinet network. For comparison purposes, the same benchmarks are also run using Berkeley UPC on the Linux cluster and using HP UPC on the AlphaServer SC cluster.

The AlphaServer SC cluster has 8 nodes with four 667MHz Alpha 21264 EV67 processors per node. The Linux cluster has 16 nodes with two Pentium 1.5GHz processors per node. The UPC compilers used are MuPC V1.1, Berkeley UPC V2.0, and HP UPC V2.2. In all experiments, each UPC thread is mapped to one processor. The cache in MuPC on each thread is configured to have $256\times$`(THREADS-1)` blocks with 1024 bytes per block. The cache in HP UPC on each thread is configured to be 4-way associative, with $256\times$`(THREADS-1)` blocks and 1024 bytes for each block.

Synthetic benchmarks results

The results for the streaming remote access tests are presented in Figures 4.2 and 4.3. These tests reveal the performance of fine grained accesses with a lightly loaded communication network. When remote reference caching is not used, MuPC has comparable performance with Berkeley UPC. Berkeley UPC is slightly better in read operations but MuPC outperforms it in write operations. It is clear that caching helps MuPC in stride-1 accesses, whose transfer rates are significantly better than the no-caching counterparts. But caching hurts the performance of random accesses, although not significantly, due to cache miss penalties. Also note that caching helps HP UPC in stride-1 reads but not in stride-1 writes. This is because the cache in HP UPC is a write-through cache. MuPC achieves only 10 – 25% of HP's performance in random accesses.

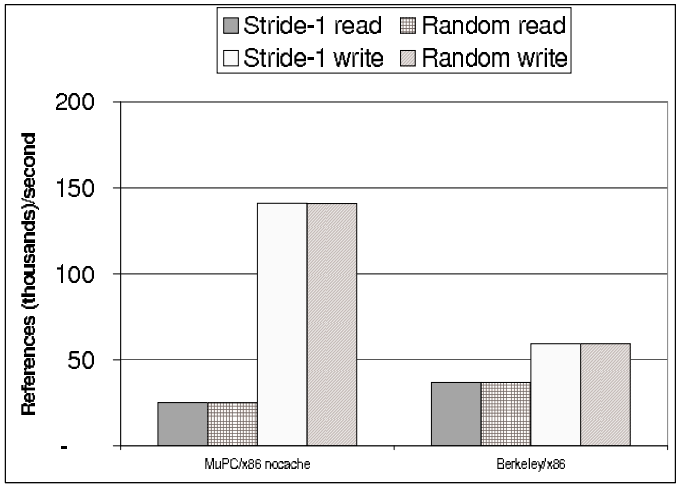

Figure 4.2: The streaming remote access results for platforms without caching

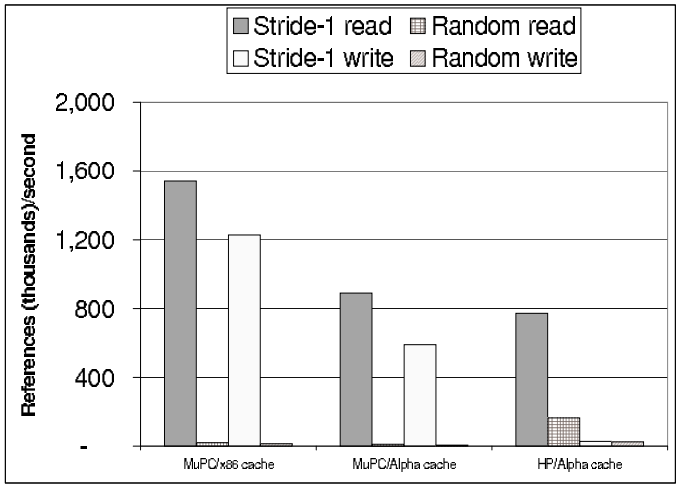

Figure 4.3: The streaming remote access results for platforms with caching

The results for the natural ring tests are presented in Figures 4.4 and 4.5. The communication network is much more heavily loaded in these tests than in the streaming remote access tests. Both MuPC and Berkeley UPC suffer from a performance degradation by a factor as large as 10. HP UPC suffers from only a slight performance degradation. The observations about remote reference caching still hold in these tests.

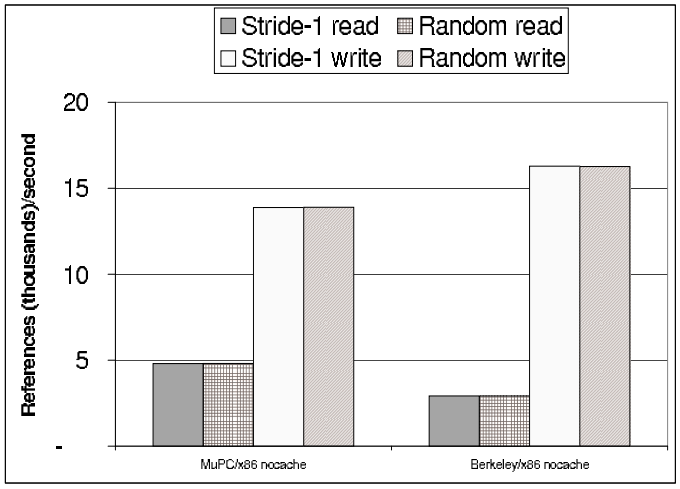

Figure 4.4: The natural ring test results for platforms without caching

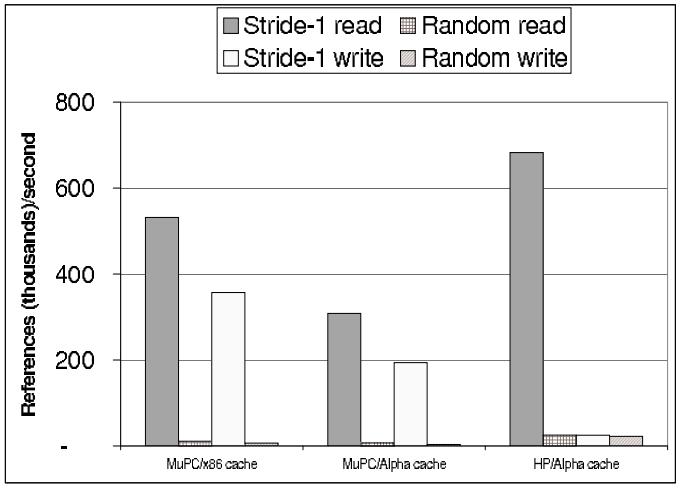

Figure 4.5: The natural ring test results for platforms with caching

NAS benchmarks results

Figures 4.6 through 4.10 display the results for the NPB 2.4 benchmarks.

CG For the nonoptimized version, this benchmark scales very poorly on all systems. MuPC performs about one order of magnitude better than Berkeley UPC on the x86 cluster and about 3 times better on the AlphaServer. HP UPC performs about 70% faster than MuPC on the AlphaServer. For the optimized version, the scalability is still poor for MuPC, HP UPC and Intrepid UPC, but the overall performance of these systems improves 5-fold. Berkeley UPC shows an order of magnitude improvement on both the x86 cluster and the AlphaServer. Berkeley UPC also achieves good scalability for the optimized version. This result shows that at the current stage Berkeley UPC is superior in handling unstructured fine-grain accesses. A cross-platform comparison shows that Intrepid UPC running on the T3E is usually slow. This is partially because the T3E machine has the slowest processors. However, since the T3E is very good at handling remote shared accesses, there should be ample room for performance improvements.

EP The embarrassingly parallel benchmark requires few remote accesses. MuPC, HP UPC and Berkeley UPC all exhibit nearly identical linear speedups on the x86 cluster and the AlphaServer.

Intrepid UPC cannot run this benchmark to completion.

FT On the x86 cluster, Berkeley UPC performs about 50% better than MuPC. They both exhibit good scalability. Optimization increases the performance by 50% for Berkeley UPC and about 100% for MuPC so MuPC and Berkeley UPC have similar performance for the optimized version. On the AlphaServer, the nonoptimized benchmark does not run to completion on Berkeley UPC. HP UPC scales poorly. MuPC has the best performance. The optimized version increases the performance for HP UPC by at least 4 times and doubles the performance for MuPC. Berkeley UPC exhibits similar performance. Intrepid UPC cannot run this benchmark to completion.

IS This benchmark scales well in all cases. On the x86 cluster, MuPC exhibits a $30-50\%$ performance edge against Berkeley UPC, for both nonoptimized and optimized implementation. Optimization doubles the performance in both cases. On the AlphaServer, HP UPC performs slightly better than others for the nonoptimized version. HP UPC, MuPC and Berkeley UPC perform very closely for the optimized version. Optimization at least doubles the performance for all three. Again, this benchmark fails Intrepid UPC on the T3E.

MG On the x86 cluster, Berkeley UPC cannot run this benchmark with less than 8 UPC threads because the problem size exhausts the available memory. When the number of threads is 8 or more, Berkeley UPC performs about 50% better than MuPC for the nonoptimized version. Both MuPC and Berkeley UPC benefit tremendously from optimization, with more than one order of magnitude improvements in performance. The two perform similarly for the optimized implementation. On the AlphaServer, HP UPC cannot run with fewer than 8 UPC threads because of memory limitations. HP UPC performs best for the nonoptimized version. Berkeley UPC performs best for the optimized version. All systems benefit by at least an order of magnitude from optimization, but HP UPC scales poorly in this case. MuPC exhibits erratic performance for the nonoptimized versions on the x86 cluster and AlphaServer because when the number of threads is 4 a large proportion of the shared accesses are local shared accesses. The performance difference between remote shared accesses and local shared accesses is obscured by the huge overall performance improvement after optimization. Intrepid UPC running on the T3E shows good scalability for both the nonoptimized and the optimized code. It benefits from optimization by more than one order of magnitude.

It has to be pointed out that the original NPB [BBB94] MPI-based benchmarks run about twice as fast as the optimized UPC-based NPB codes on all three platforms. Part of this performance difference may be attributed to the maturity of UPC compilers and run time systems. Another part of this difference may be due to MPI's fitness for the NAS benchmarks, though not all of them are coarse-grained. The benchmarks used in this work were not written "from scratch" in the UPC style but were translated from the MPI versions to UPC, so they do not necessarily take advantage of the asynchronousness natural to UPC's relaxed memory access mode but are more attuned to the bulk synchronous style of MPI. Current UPC compilers do not exploit the greatest part of the optimizations available by instruction reordering offered by UPC's relaxed memory access mode. Finally, distributed memory platforms such as the x86 cluster and the SC-40 were not designed to minimize the costs of fine-grained accesses. Needed to resolve this question is further development of compilers and run time systems, development of benchmarks suitable for fine grained computations, and new parallel architectures to facilitate the investigation of the partitioned shared memory programming model.

4.7 Summary

MuPC is an open source runtime system for UPC. It is based on MPI and POSIX threads and therefore is portable to most distributed memory and shared memory parallel systems. It consists

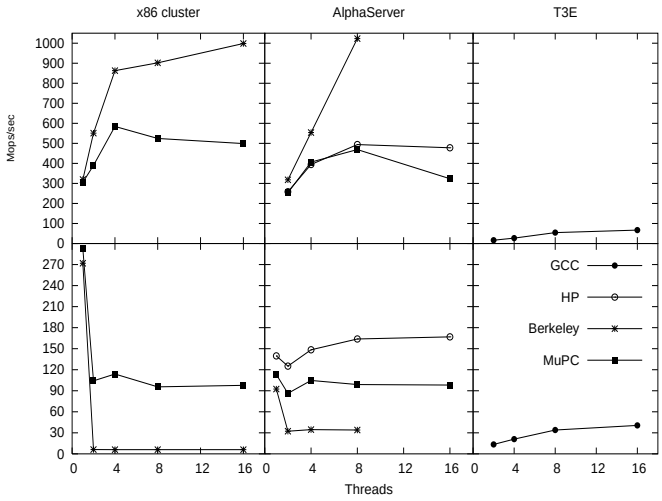

Figure 4.6: CG benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version. Note: Measurements for Berkeley UPC on the AlphaServer for the nonoptimized version were collected using version 2.0.

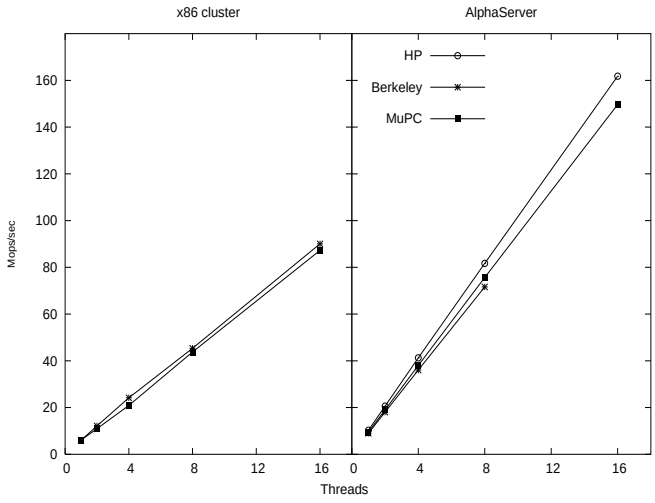

Figure 4.7: EP benchmark performance. This benchmark failed Intrepid UPC. Note: Measurements for Berkeley UPC on AlphaServer were collected using version 2.0.

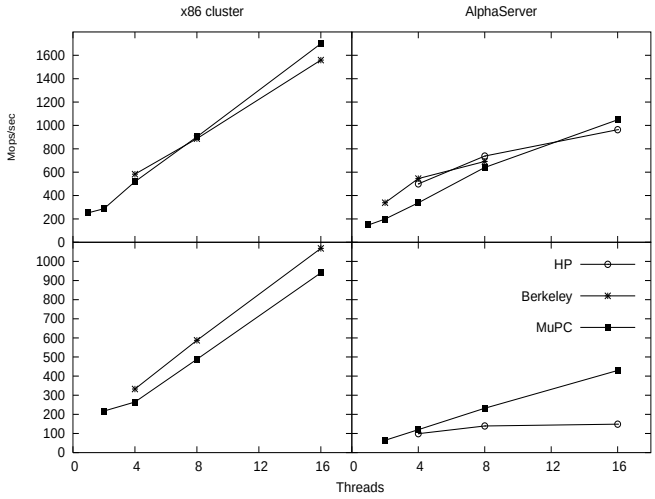

Figure 4.8: FT benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version. This benchmark failed Intrepid UPC. The nonoptimized version also failed with Berkeley UPC on the AlphaServer. Note: Measurements for Berkeley UPC on the AlphaServer were collected using version 2.0.

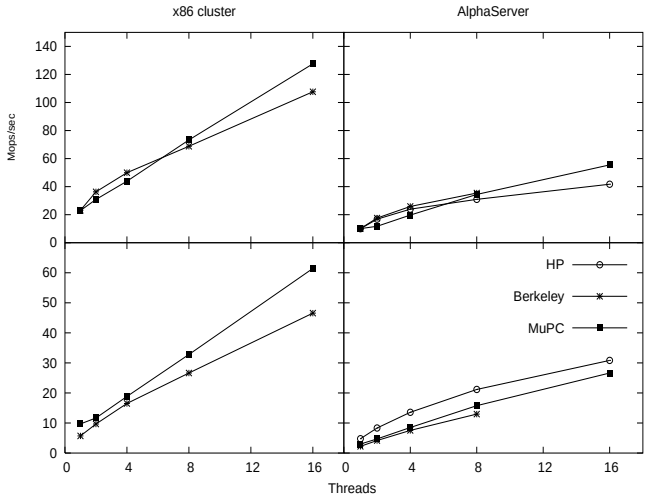

Figure 4.9: IS benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version. This benchmark failed Intrepid UPC. Note: Measurements for Berkeley UPC on the AlphaServer for the nonoptimized version were collected using version 2.0.

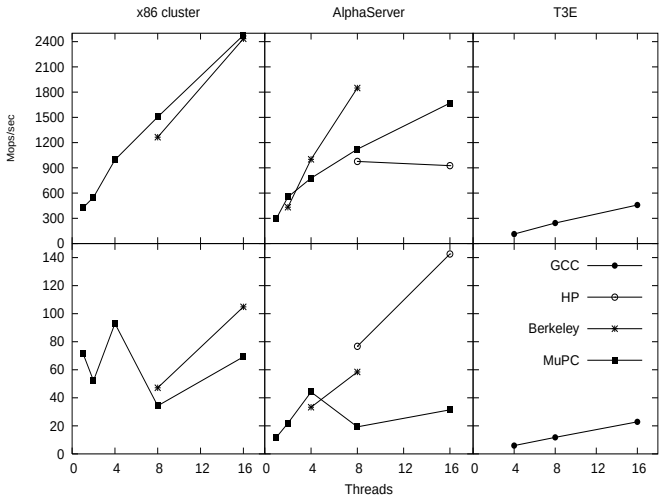

Figure 4.10: MG benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version.

of a runtime library, a UPC-to-C translator, and a collective function library. This chapter described the internals of the runtime library, which is the core of MuPC.

The UPC-to-C translator works as a front-end that translate UPC code into ANSI C code. A set of UPC runtime API functions provide an interface between the translated code and the runtime library. An optimization performed at the UPC source level is localizing shared references with local affinity. These references are translated into regular memory accesses, bypassing the runtime library. Other optimizations include a software cache fore remote accesses.

Each UPC thread is represented by two Pthreads at run time; one for computation and one for inter-thread communication. The computation Pthread executes instructions in a ANSI C program, which is translated from a UPC program. Accesses to non-local shared memory locations are transformed to MPI messages and are dispatched to the communication Pthread for processing.

Most techniques used in MuPC implementation can also be applied to the implementations of other PGAS languages. Since PGAS languages have similar shared memory abstraction and execution model, they can share the same runtime system. In principle, for a PGAS language, as long as there exists a translator that translates source code into C code then the MuPC runtime system can be used as the runtime system for this language.

Performance results in this chapter compare several current UPC compilers with synthetic and application benchmarks, showing that MuPC achieved comparable performance with other UPC implementations. In the following chapters dedicated to performance modeling MuPC faciliates more performance benchmark development. MuPC together with other UPC implementations are also used to validate the performance modeling results.

Chapter 5

A study of UPC remote reference cache

The global address space of most UPC platforms is implemented as a software layer on the top of distributed memory. This layer creates a "memory wall" that is orders of magnitude larger than the one found between regular memory systems and CPUs. Caching is a natural choice for hiding the very long latency of remote shared accesses. So far, however, only MuPC and HP-UPC have implemented remote reference caching mechanisms. There are probably three reasons why other UPC implementations do not do the same.

First, direct hardware support for remote access caching is uncommon on most distributed memory systems. UPC implementations must use software caches. The overhead of a software cache is much larger than that of a hardware cache. There has never been a thorough study of the effectiveness of any of the current UPC cache systems to justify this overhead. Second, a cache is just a way to increase memory throughput. It in no ways improves the actual memory latency. The ability of a cache to increase memory throughput heavily depends on reference patterns of applications. The UPC programming model makes it easy to write programs with irregular reference patterns, which are typically cache unfriendly. Many UPC implementers therefore would rather focus on reducing the inherent shared memory latency, or reducing the number of remote accesses with compiler optimization techniques. Third, cache coherence has always been a hurdle in distributed and parallel computing. The cost of enforcing cache coherence for a UPC software cache is prohibitive. The UPC memory model does allow cache incoherence under the *relaxed consistency mode* (both MuPC and HP-UPC provide non-coherent cache), but it requires non-trivial efforts from programmers to understand this memory model. So even when a cache is provided, it is considered to be an advanced feature that only expert-level UPC programmers can safely handle.

Nonetheless, among all productivity-improving techniques, remote reference caching is the easiest and cheapest to implement. Its performance deserves a careful investigation from the performance perspective to quantify its effectiveness and to illustrate the trade-offs in cache design. There are two objectives in such an investigation: To explain the cache effects not accounted for in the run time prediction using the performance model given in Section 6, and to provide insights for choosing good cache parameters for UPC programs with various reference patterns.

This chapter uses a synthetic global data access benchmark to investigate the interactions between shared memory access characteristics (namely spatial locality and temporal locality) and cache configurations. Important discussions are given about designing cache-friendly programs and choosing optimal cache configurations based on an application's shared memory access characteristics.

So far, performance measurements reported in this report for cache-enabled MuPC and HP-UPC were obtained using default cache parameters. For example, in HP-UPC cache block size is 1024 bytes, the number of cache sets is 256, and cache associativity is 4. There is no particular justifica-

tion for these settings because little is known about how cache configuration affects the performance of applications with different reference patterns.

Cache performance analysis is much more difficult than building the software cache itself. Traditional approaches rely on simulations and execution trace analyses. These approaches usually use benchmark suites that contain real-world applications. The results are applicable only to applications with similar characteristics. UPC is still in its early adoption stage. There is not yet a mature and large code base from which application benchmark suites can be built. Tools for simulation and trace generation are still in early development stage [SLB+06]. To overcome these difficulties, this study uses the Apex-MAP synthetic benchmark [SS05, SS04] developed by the Future Technology Group at the Lawrence Berkeley National Lab. Apex-MAP is a synthetic global data access benchmark for scientific computing. It can generate performance probes for the whole range of spatial and temporal locality for any UPC platforms [SS05]. In this section, Apex-MAP is used to analyze remote reference caching in MuPC and HP-UPC. These results and methodology can be a guide for selecting optimal cache parameters based on an application's remote reference patterns.

5.1 An overview of MuPC cache and HP-UPC cache

The remote reference cache in MuPC is direct-mapped. A small fully associative victim cache is used to reduce conflict misses. Remote writes are also cached (write-back), but coherence is not maintained. A more detailed description of the caching mechanism can be found in Section 4.4. Cache parameters are set in the user's MuPC configuration file (`mupc.conf`).

The cache in HP-UPC supports remote reads only. Writes are not cached. Its associativity is tunable. There is no victim cache. Cache parameters are set by using environment variables.

The following are the specifications for the two caches:

	MuPC	HP-UPC
Block size	64–2048 bytes	64–8192 bytes
Cache sets	16–1024 $\times$ `THREADS`	Unlimited
Associativity	direct-mapped	1–128

5.2 Apex-MAP

Apex-MAP is an architecture independent memory performance benchmark. It is based on the assumption that the memory access patterns of scientific applications can be abstracted using three memory hardware independent factors: data set size, spatial locality, and temporal locality. The first two factors are easy to quantify. The temporal locality is determined by the re-use distance, which is approximated using the power distribution function. In other words, the addresses of a sequence of memory accesses comply with the distribution $A = a^{\frac{1}{K}}$, where a is a random number and K is the temporal locality quantifying parameter. Choosing the power distribution function to approximate temporal re-use is appropriate because the distribution is scale-invariant. A single parameter therefore is enough to characterize temporal locality without regard to the memory size.

Apex-MAP defines three parameters in total:

- **M** is the data set size, which is the total amount of memory that may be accessed in one execution.

- **L** is the vector size (double words) of an access. This parameter determines the spatial locality. The larger L is, the more double words are read in one access sequence, and the more spatial locality is exhibited.

- **K** is the shape parameter in the power distribution function. It is a value between 0 and 1. The lower the value, the more temporal locality is exhibited. $K = 1$ means a random access pattern, and $K = 0$ means repeated accesses to one location.

Apex-MAP was designed to probe the performance of the whole memory system. To accommodate the special needs in studying remote reference caching in UPC, this work uses a modified version of the original Apex-MAP program. The first and the most important modification is the exclusion of local shared accesses. The original Apex-MAP does not differentiate between the remote and local shared accesses. It happens to have an unfortunate property that increasing the temporal re-use (a lower K value) also increases the number of local shared accesses. Since both of the UPC software caches under investigation are designed only for remote accesses, continuously increasing the local accesses would obscure the analysis. The Apex-MAP code is modified so that the overwhelming majority of the accesses made during benchmarking are to remote locations. This modification also makes sure that the shared memory locations accessed by remote references are balanced among participating threads: No locations are accessed by multiple threads at the same time. This ensures the probes do not generate hot-spots in the shared memory, and the effects of memory contention are not going to interfere with the observations. The second modification deals with result reporting. The original benchmark reports an accumulated bandwidth, which is defined to be the sum of the bandwidths of all threads. But this study is only concerned about the effectiveness of the remote reference caches, which is independent of the number of threads since the software cache is replicated on each thread. Therefore, the modified Apex-MAP reports an average per-thread bandwidth.

Only read accesses with unit stride are measured by the benchmark. This is the most common type of remote access in UPC applications.

5.3 Experiments and results

Experiments were conducted using MuPC on a Linux cluster with a Myrinet interconnect, and HP-UPC on a AlphaServer SC-40 SMP cluster. The size of shared memory probed (M) is constant during measurements and is at least 32 times larger than the cache size. Four factors are carefully controlled during the experiments. Parameters L and K are used to control spatial and temporal locality. Cache block size and the number of cache sets per thread are used to control cache geometry. These factors range over the values given in the following table.

Factor	Levels
L	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
K	1.0, 0.5, 0.25, 0.1, 0.05, 0.025, 0.01, 0.005, 0.0025, 0.001
Block size	64, 128, 256, 512, 1024, 2048 (bytes)
Cache sets per thread	16, 32, 64, 128, 256, 512, 1024

Each combination of the four factors is benchmarked, resulting in $11 \times 10 \times 6 \times 7 = 4620$ data points on each platforms. Each point is measured at least 16 times and the average is taken. All experiments are designed to avoid interference from shared memory contention. The number of

cache sets in both MuPC and HP-UPC scales with the number of threads (See Section 4.4). So the number of UPC threads is not a factor contributing to bandwidth readings.

Measurement results are plotted against the four factors respectively, giving the whole picture of caching effectiveness from multiple perspectives. In all plots the logarithmic values (base 10) of observed bandwidths are actually used because the difference between the worst and the best readings is more than two orders of magnitude.

Plots obtained for MuPC cache and for HP-UPC cache are very similar. In the following discussions only MuPC cache plots are displayed. But the conclusions apply to HP-UPC as well. Associative cache is a feature specific to HP-UPC, so it is discussed separately in Section 5.3.5.

5.3.1 The *L*-*K* perspective

The readings are plotted against *L*-*K* combinations. Selected curves are presented in Figure 5.1. In each of the six plots, there are 11 groups of curves. Each group is corresponding to a certain L value. Within each group 10 data points are plotted for each curve, corresponding to 10 K values. For example, the group labeled in the X-axis with "32-[1:0.001]" were obtained using $L = 32$(double words) and K varying from 1.0 to 0.001. Each group has four curves, corresponding to four cache settings: A fixed cache block size with 16, 64, 256, and 1024 cache sets, respectively. The six plots are different only in the cache block size, which are fixed at 64, 128, 256, 512, 1024, and 2048, respectively in each of these plots.

Remote reference caching improves memory throughput significantly in most cases. The average bandwidth without caching, as measured by the *Baseline* microbenchmark described in 6.4, is 0.16 MB/sec. The base 10 logarithmic value is -0.80. Only a very small number of data points in the plots are below this level. Those few data points are located in the lower left corner in each plot, corresponding to random access patterns ($K = 0.25$) with the shortest access vector ($L = 1$). In all other cases caching yields improvements of 50% to more than two orders of magnitude.

The four curves in each *L*-*K* group almost fully overlap. This is a strong indication that the number of cache sets has little effect on remote memory bandwidth and it is independent of other factors. There is only one exception: In the first plot when L is 1024 the curves for small numbers of cache sets (16 and 64) are much flatter than the other two. The two low performance curves are results for very small caches with only 64-byte cache blocks and 16 or 64 cache sets. When large vectors are accessed ($L = 1024$) with non-negligible temporal reuse ($K \leq 0.1$), too many cache misses due to capacity conflicts inhibit the bandwidths.

Among the 11 groups in each plot, those on the left side tend to have a greater span than those on the right side. This means that bandwidth is more sensitive to temporal locality when L is small than it is when L is large.

Going from left to right and top to bottom, cache block size doubles for each graph but bandwidths for high temporal locality (the upper ends of curves) remains roughly the same across the six graphs. Bandwidths for low temporal locality (the lower ends of curves) improve significantly with the doubling of cache block size, but most likely this is the contribution of the increasing of L factor.

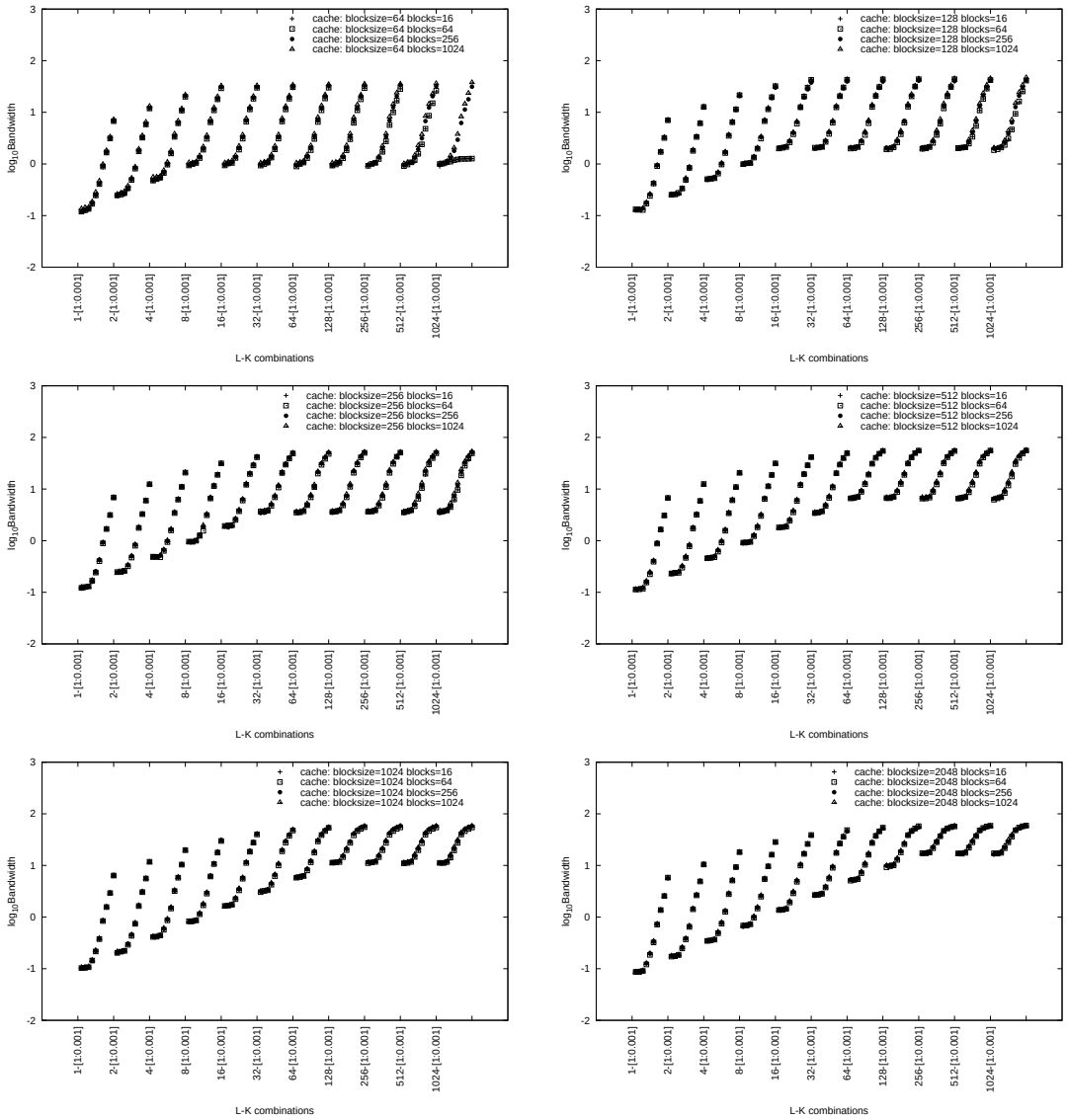

Figure 5.1: Logarithmic bandwidths against *L*-*K* combinations

5.3.2 The *K*-*L* perspective

The *K*-*L* view presented in Figure 5.2 are plots of exactly the same set of data as plotted in the *L*-*K* view. The difference is that data points are categorized using *K* values this time. There are 10 groups of curves in each graph, corresponding to 10 *K* values benchmarked. Within each group are points corresponding to the 11 *L* values. The settings for cache block size and the number of cache sets in all six plots are the same as before.

There is a plateau on each curve. The plateaus are longer for smaller cache block size and shorter for larger cache block size. Within each graph the plateaus are of the same length for all groups. The plateaus are because of the saturation of the cache. Small caches are saturated easily with small *L* values (so they have longer plateaus) and large caches can accommodate much larger *L* values.

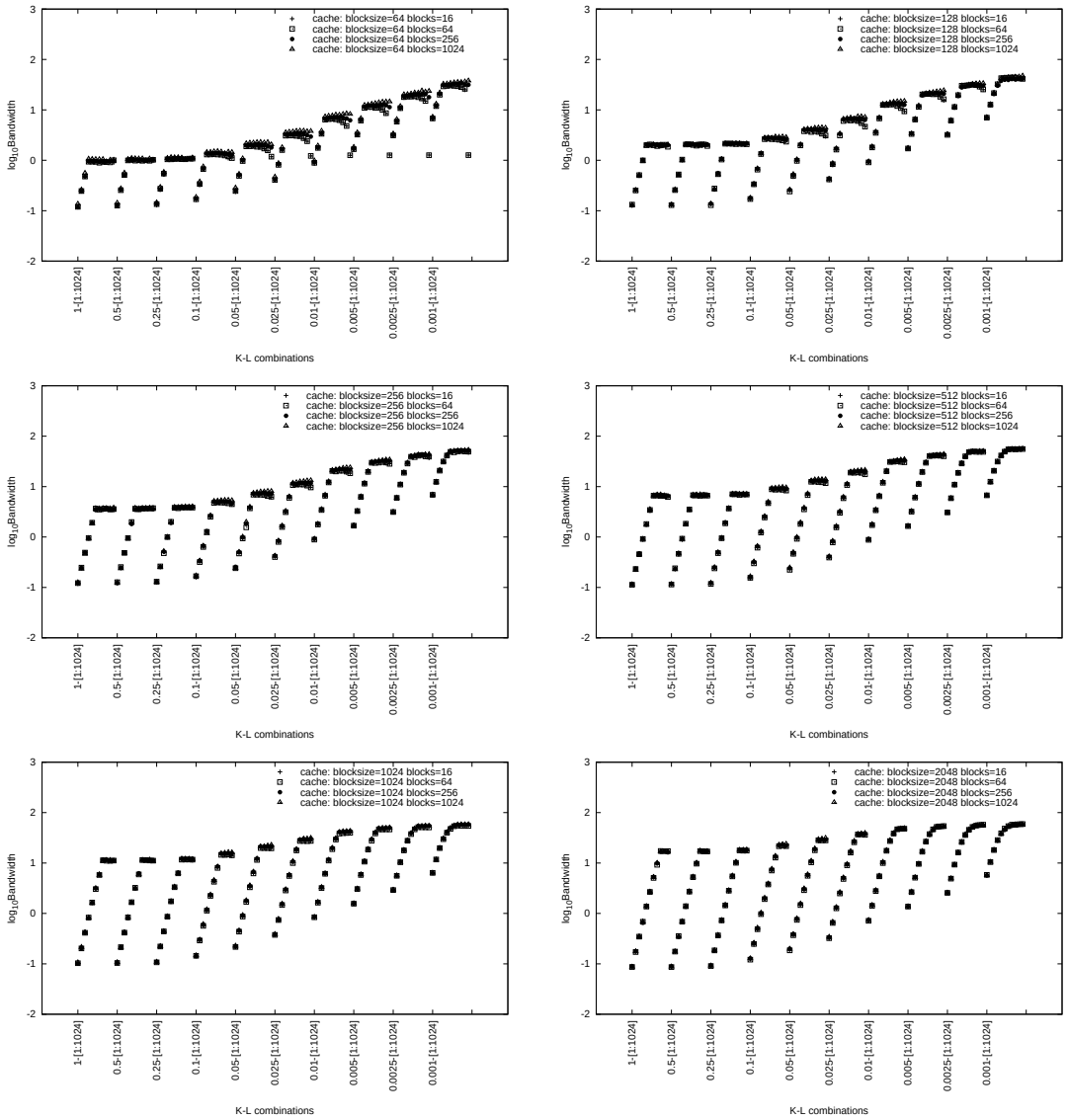

Figure 5.2: Logarithmic bandwidths against *K*-*L* combinations

The heights of the plateaus reflect the best bandwidths of various reference patterns (*K*-*L* combinations) under different cache settings. Larger cache block sizes lead to higher plateaus. On the other hand, the opposite ends of the plateaus, corresponding to low spatial locality, do not change with the increasing of cache block size. Therefore, accesses with high levels spatial reuse need large cache block size, while accesses with little spatial reuse benefit little from large cache blocks.

The number of cache sets does not matter, as observed before.

5.3.3 The cache block size perspective

Figure 5.3 was obtained by plotting the logarithmic values of bandwidth measurements against cache geometry. Data points are categorized using cache block sizes. There is one graph for each of the six L values. Going from left to right and top to bottom, the L value quadruples for each graph. Four curves are plotted each graph, corresponding to 4 levels of temporal locality ($K = 1$, $K = 0.1$, $K = 0.01$, and $K = 0.001$).

In each graph, access patterns with $K > 0.1$ show little difference in their performance. Their curves appear at the bottom. Performance improvements achieved when $K \leq 0.01$. Across graphs, access patterns with no spatial reuse ($L = 1$) show the lowest bandwidths. As long as there is a little spatial reuse ($L = 4$), the bandwidths improve about one order of magnitude. But the improvements due to L are very limited after this; the positions of the four curves on the $L = 64$ graph are almost the same with those on the $L = 1024$ graph.

Long cache blocks bring significant improvements only for cases where temporal reuse is poor (K close to 1) but spatial reuse is strong ($L > 64$). When L is not large, however, all four curves on each graph are almost flat. This proves that temporal locality (the K value) is insensitive to cache geometry. Given large enough cache blocks, accesses with substantial spatial locality can achieve good bandwidths regardless of their temporal reuse characteristics.

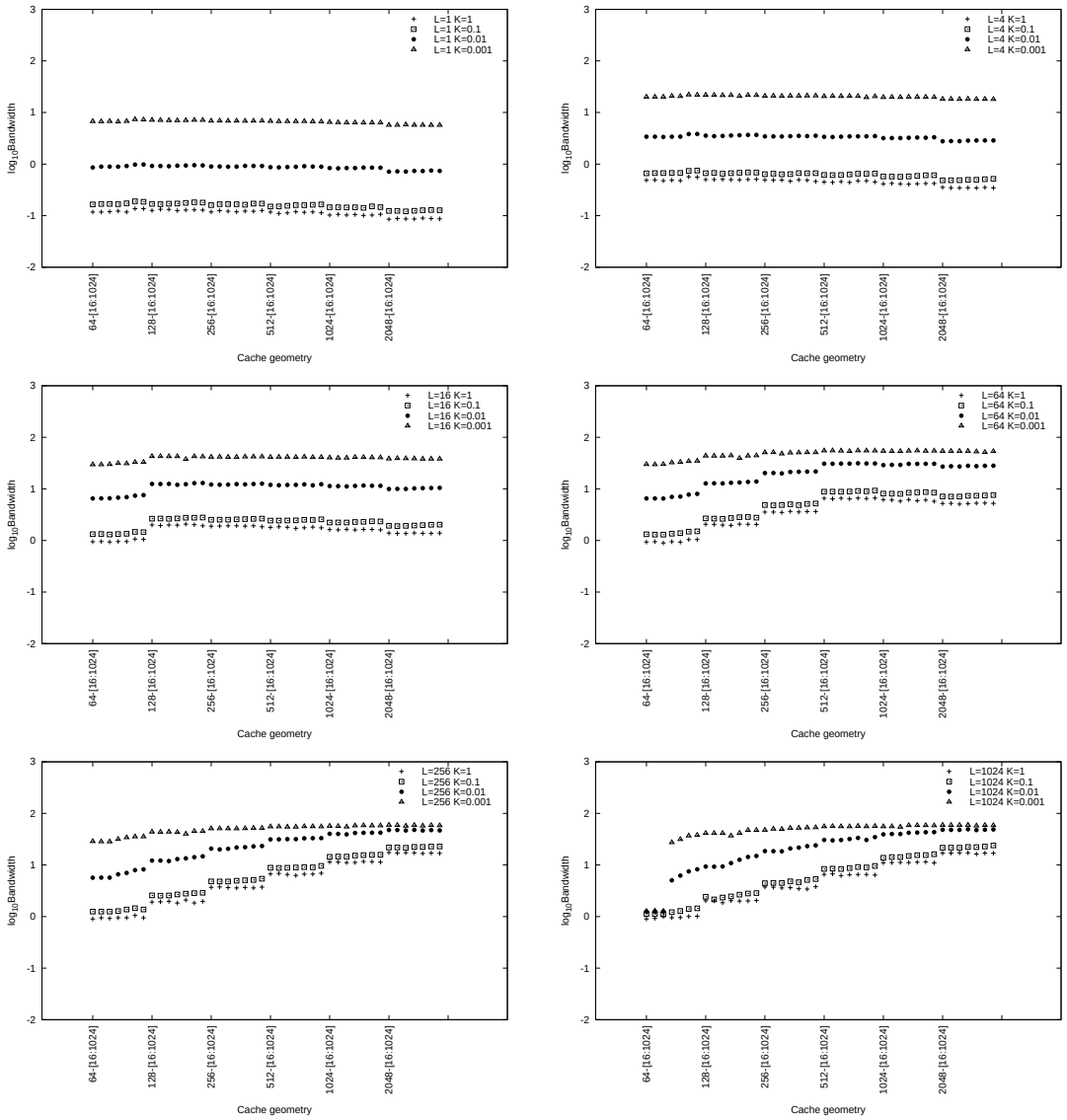

Figure 5.3: Logarithmic bandwidths against cache blocksize and set numbers

5.3.4 The cache table size perspective

Categorizing the same set of data points using the number of cache sets per thread yields Figure 5.4. There are 7 groups of curves on each graph corresponding to the 7 levels of cache table size. The six graphs are corresponding to the six L levels, and the same four levels of K are plotted on each graph.

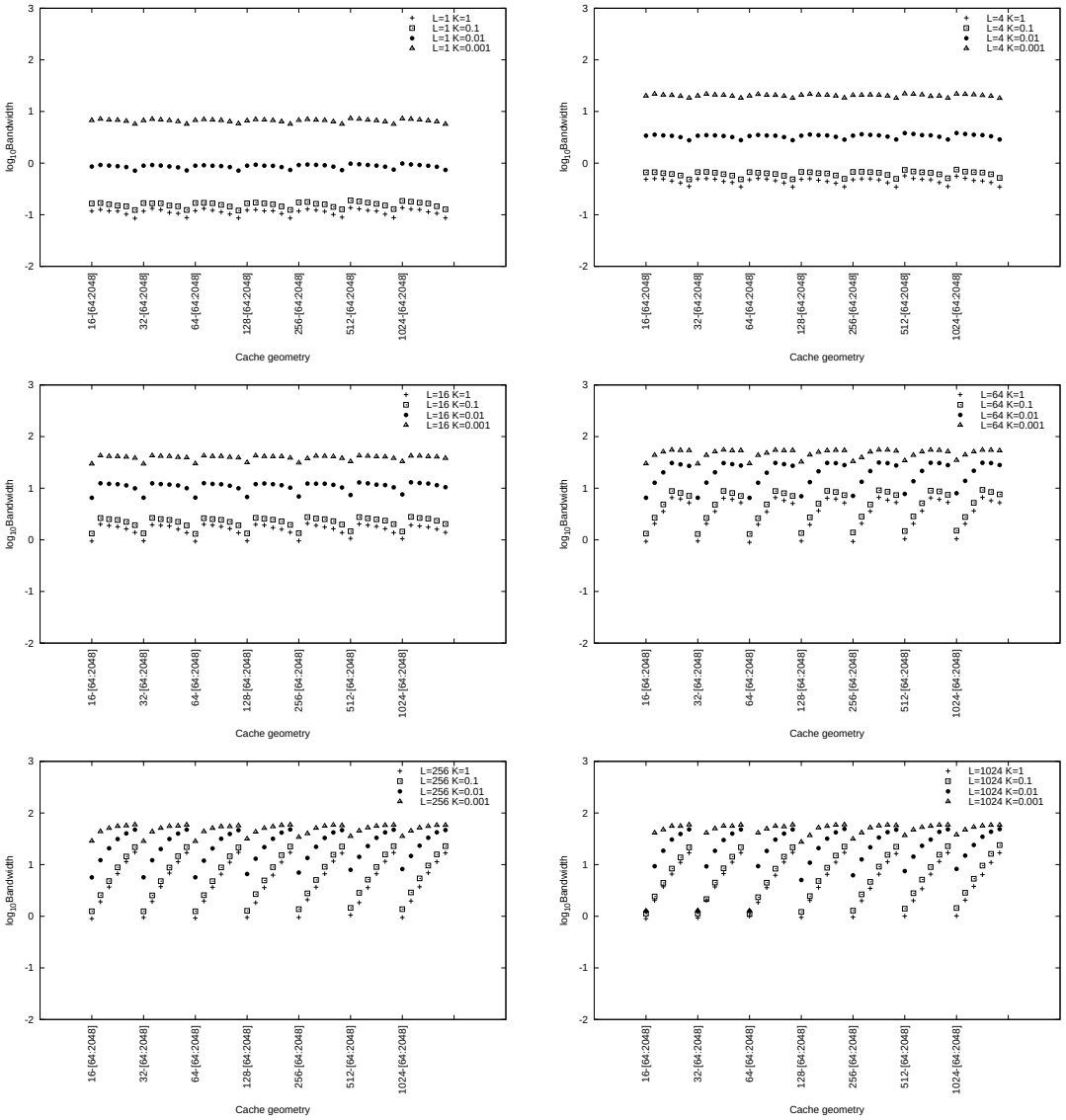

Figure 5.4: Logarithmic bandwidths against cache set numbers and blocksize

The shapes of curves are different among the six graphs. However, the shapes of the 7 groups in each graph are the same. This means cache table size per thread is a negligible factor in determining bandwidth.

When $L \leq 16$, increasing the cache block size has no benefits. It even slightly degrades the bandwidths for low temporal reuses. As L grows, increasing the cache block size boosts the bandwidth, especially for accesses with low temporal reuse. But the benefit brought by increasing the cache block size is not long-lasting; it stops when the cache block size grows beyond the value of L.

5.3.5 Cache associativity

All discussions above apply to direct-mapped cache. HP-UPC (V2.3) provides associative cache. The associativity is tunable at run time through an environment variable. Associative cache is usually used to improve the hit rate by reducing conflict misses. In HP-UPC (same with MuPC) every thread has a designated chunk in the cache table, so memory locations on different threads are mapped to separate chunks. This avoids cache line conflicts among threads and largely offsets the benefits of an associative cache. Figure 5.5 and Figure 5.6 clearly confirm this conclusion. Figure 5.5 shows bandwidth measurement results for a direct-mapped cache and Figure 5.6 for a 8-way associative cache. The curves between corresponding settings are strikingly similar, indicating that associativity has little effect on hit rate.

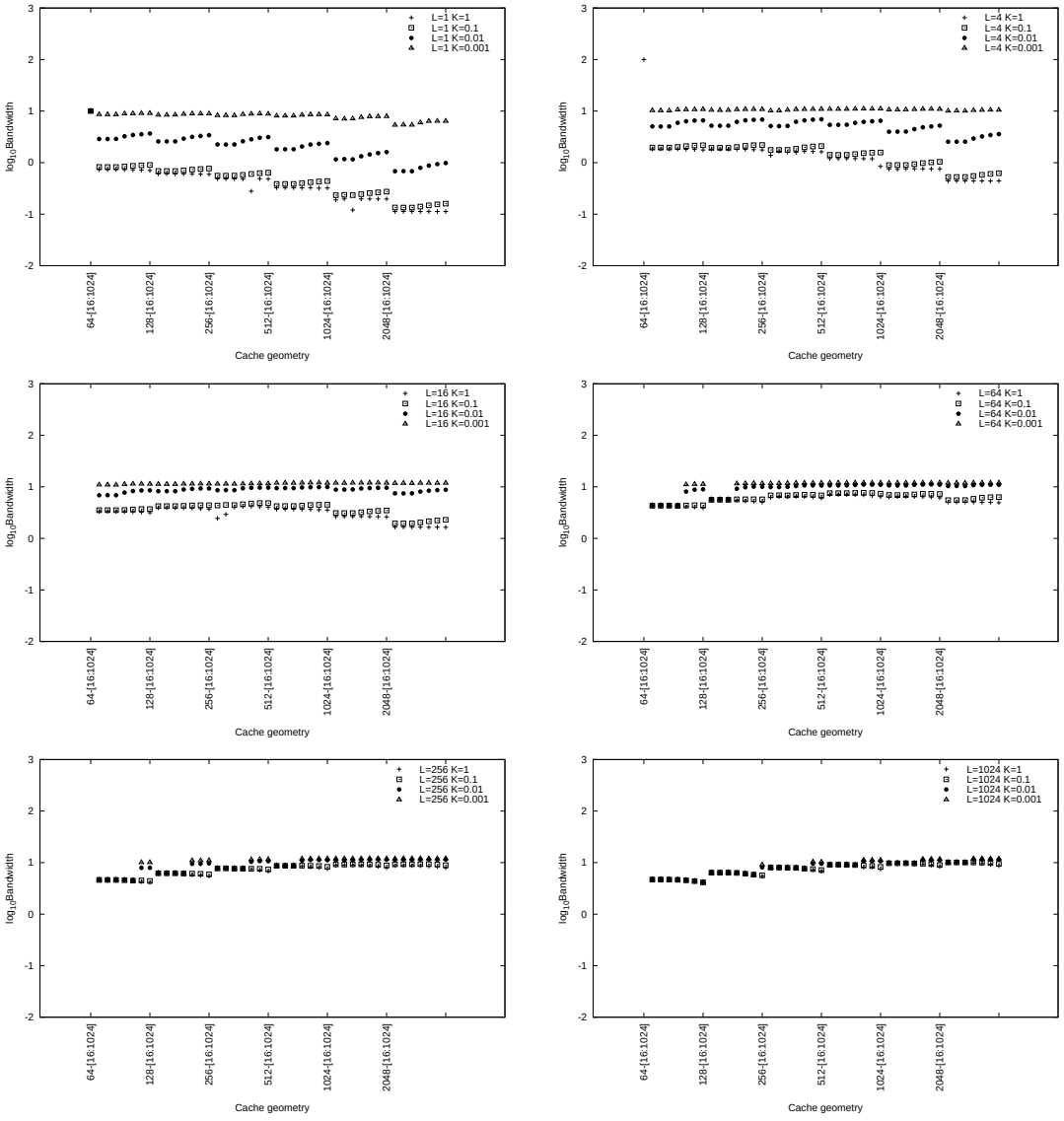

Figure 5.5: Logarithmic bandwidths: HP-UPC with direct-mapped cache

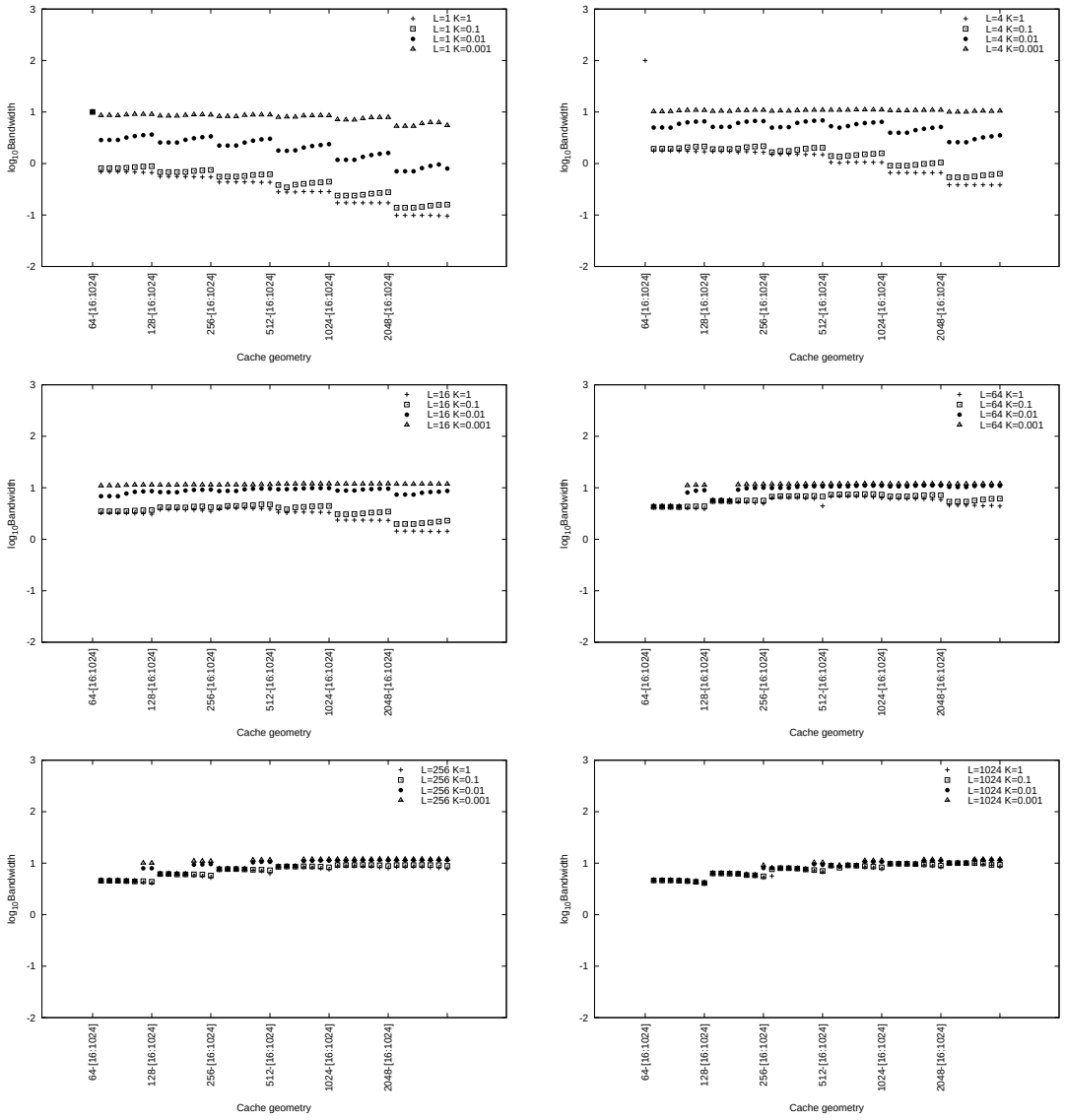

Figure 5.6: Logarithmic bandwidths: HP-UPC with 8-way associative cache

5.4 Conclusions

Several important conclusions can be drawn from the Apex-MAP benchmarking experiments.

1. Overall, remote reference caching improves remote memory throughput significantly. If an application performs a large number of remote reads and little remote writes, then it is always beneficial to use caching. The benefits of caching in applications with lots of remote writes is more problematic. Caching may have negative effects because of the cost of frequent cache invalidation when the writes exhibit poor temporal locality.

2. The number of cache sets per thread is a negligible factor. By design the MuPC cache supports up to 1024 cache sets per thread, but there is little evidence to justify more than 64 cache sets per thread. However, it must be noted that this conclusion only applies to programs whose remote accesses are nearly all remote reads. This is because the MuPC cache caches both

reads and writes, and conflict misses are much more expensive for writes than for reads. The write miss penalty includes the cost of moving a whole dirty block back to the memory.

3. The length of a cache block is a significant factor. Cache blocks with less than 64 bytes are not very useful. Long cache blocks are needed for long vector accesses, but cache blocks longer than the vector length are unnecessary.

4. Temporal locality is a very significant factor affecting the remote memory bandwidth. Applications with high levels of temporal reuses easily gain orders of magnitude bandwidth improvements with caching.

5. Spatial locality is easy to exploit with large cache blocks. Since temporal reuse is a special form of spatial reuse, applications that exhibit little temporal reuse may still exhibit spatial reuse. One example are the applications that make random but clustered remote accesses. No temporal reuse is observed in these applications. But if we treat a whole cache block as a reference unit then the cost of each individual access in the block is amortized due to spatial reuse. Therefore, setting large cache block size always improves the performance of this type of applications.

6. Cache associativity is virtually useless in improving hit rate. A direct mapped cache works just as well as an 8-way associative cache. Actually, from V2.4 on HP-UPC has removed this feature, keeping only a simple direct mapped cache.

These conclusions have important implications on how to design cache-friendly UPC programs, as well as on how to tune cache settings at run time to achieve optimal performance based on a program's remote access pattern.

A well-designed UPC program should try to minimize the need for remote data. Then, for those inevitable remote accesses it should maximize temporal reuses. If this is not possible then it should perform clustered remote accesses (so spatial reuse is high) as much as possible and then use large cache blocks to exploit the benefits. Programs with many remote writes should consider using large cache tables to mitigate conflict miss penalties.

Theoretically, at run time an optimal cache configuration can be set based on the memory access characteristics of the program to be run, because the Apex-MAP benchmarking results cover all combinations of a wide range of temporal and spatial locality. In practice, however, the correlation between a program's temporal locality and the L and K parameters defined in Apex-MAP is not always obvious. One limitation of the Apex-MAP tool is that it does not provide an easy way to deduce the corresponding L and K values for a given access pattern. A brute force method is to use a statistical back-fitting procedure to find the parameters for a given program [SS04]. The program is run with various data set sizes. A large $L-K$ space is then explored using Apex-MAP, with the sizes of probed memory being set to according program data set sizes. The run times obtained from running the program are statistically compared with those produced by Apex-MAP under various $L-K$ combinations, to find the exact values of L and K.

These procedures are too complicated for practical uses. For application kernels with simple access patterns the exact values of L and K are not important. Qualitative analysis is usually good enough to provide guidelines for choosing optimal cache settings. The example in the next section illustrates this point.

5.5 Example

The UPC matrix multiply kernel has a very regular access pattern. The naïve $O(N^3)$ implementation discussed in Section 7.2 with cyclic striped distribution of input matrices exhibits very strong spatial reuse because remote columns are accessed as vectors. The remote columns are accessed in a round-robin fashion so the temporal distance is very long, implying poor temporal locality.

Based on the earlier discussion, the rule of improving performance for this type of access patterns is to use a cache with long cache blocks to take advantages of the vector accesses. But there is another alternative. Since temporal locality is a more significant factor than spatial locality in affecting remote access performance, using a different implementation that is targeted to exploit temporal reuse should be more effective than relying solely on the cache.

The matrix multiply kernel is re-written to take the following form:

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

// Initialize matrix C to all 0's.

for(j=0; j<N; j++) {
  for (k=0; k<N; k++) {
    upc_forall(i=0; i<N; i++; &A[i][0]) {
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
upc_barrier;
```

The `upc_forall` loop is moved into the innermost level but still distributes work based on the affinities of the rows of matrix A. In this implementation the local rows of matrix A are accessed in a round-robin fashion. The remote columns of matrix B are now accessed in a way such that each column is used repeatedly to multiply with all the rows in A before going to the next column. Table 5.1 show the run times and cache miss rates of running the two implementations using different sizes of cache blocks.

	Cache block size (byte)								
	no cache	256		512		1024		2048	
	second	second	miss rate	second	miss rate	second	miss rate	second	miss rate
naïve	415.86	297.96	56.08%	115.83	19.48%	5.67	0.03%	4.16	0.01%
temporal	437.66	10.88	1.14%	7.64	0.36%	6.05	0.03%	5.93	0.01%

Table 5.1: Running two implementations of the matrix multiply kernel using MuPC remote reference cache. The *naïve* code exploits remote access spatial reuse and the *temporal* code exploits remote access temporal reuse. Input matrices are 400×400 in dimension with floating point type elements. The number of cache sets used in all experiments is fixed at 64.

The results confirm with the previous expectation. Without remote reference caching both implementations take about the same amount of time. But when using a cache with only small cache

blocks (256 bytes), the *temporal* implementation run time drops by more than one order of magnitude. The cache miss rate is only 1.14% while the *naïve* code has more than a 50% miss rate. The run time and the miss rate for both continue to fall as the cache block size doubles. Once the cache block size reaches 1024 bytes and above, the two implementation have virtually no difference in terms of the cache miss rate. This is because at this time the cache capacity is so big that a substantial portion of an input matrix can reside in the cache without being evicted due to capacity conflicts, even when the columns of the matrix are accessed with long temporal distance.

Note that with long cache block sizes the *temporal* implementation takes more time to finish than the *naïve* implementation, though their miss rates are the same. The explanation is, once the most part of the remote access cost is mitigated by caching, the cost of local accesses becomes important. The column-major accesses of matrix A are unfriendly to the CPU cache, thus increases the total run time.

Chapter 6

Performance modeling for PGAS languages

While substantial efforts have gone into PGAS language designs and implementations, little work has been done to model the performance of these languages. At the first PGAS Programming Models Conference [PGA05] and at the 5th UPC Developers Workshop [Geo05], both held at the end of 2005, the need for a performance modeling methodology for PGAS languages, especially for UPC, was expressed many times by application developers and language developers.

This work is the first to address the demand for an application level performance model for PGAS languages. UPC is chosen to be the representative of PGAS languages and the target for performance model validation.

This study focuses on the performance of fine-grain shared memory accesses in an execution environment with relaxed memory consistency. This chapter first investigates the UPC programming model to isolate the inherent factors that complicate performance modeling. Then an approach is proposed to describe the interactions between a UPC platform and UPC applications. The UPC platform is abstracted into a small set of features and then these features are mapped onto the shared references in an application to give a performance prediction. Microbenchmarks are given to quantify the set of UPC platform features, and the principles of a dependence-based algorithm to characterize shared references are also given. Finally, the performance model is validated using three simple UPC programs.

This chapter starts with a review of important performance models for message passing and shared memory communications. Then, a few features of the UPC programming model that have implications to UPC performance will be reviewed to explain why they defeat efforts of modeling UPC performance using existing models. This section is followed by discussions on platform abstraction and application analysis techniques used in this work. Next, microbenchmarks designed for characterizing a UPC platform are discussed in detail, followed by mathmetical equations to predict run time and characterize concurrency. At the end of this chapter there is a discussion about extending the performance modeling methodology to other PGAS languages, such as Co-Array Fortran and Titanium.

6.1 Performance modeling of parallel computing

Over the past decade many performance models for parallel computing have been proposed. Since MPI and clusters are so widely used, the majority of these performance models target point-to-

point communication and are based on network properties and message sizes. Models of this kind include BSP [GV94] and LogP [CKP+93], and many variations [DDH97, HK96, AISS95, IFH01, MF98, CS03, HHS+95]. The "check-in, check-out" (CICO) performance model [LCW93] has been proposed for the shared memory programming model but it is only applicable to cache-coherent architectures. None of these models are suitable for UPC because mechanisms employed by UPC, such as the global address space, fine-grain references, and implicit communication, are not captured by these models.

6.1.1 Models for point-to-point communication

The LogP model [CKP+93] was developed in the early 90's. It defines a simple abstraction of network interconnects found in most modern parallel machines, yet it is accurate enough to predict the communication cost for message-passing parallel programs. The LogP model captures the most important details of an interconnect using four parameters: L: An upper bound on the cost of one-word message transfer latency; o: The amount of time a processor is tied up in handling messages and cannot do other work; g: The reciprocal of the effective per-processor communication bandwidth; and P: The number of processor/memory module. The cost of a point-to-point message transfer is charged with cost: $o_s + w \times \max\{g,o\} + L + o_r$, where w is the message size in words. The key point of the LogP model is that it suppresses system specific information including network topology, message buffering, routing algorithms, local memory interaction, and so on. All of these are distilled down to the L, o, and g parameters.

The simplicity of the LogP model makes it less accurate for systems where message size, synchronization, network contention, memory hierarchy and middleware costs substantially affect performance. To capture these effects, a variety of extensions have been proposed, each dealing with one of these aspects. For example, the LogGP model [AISS95] extends LogP with an additional parameter G, which accounts for the cost of long messages. A w-word long message is then charged with $o_s + w \times G + L + o_r$. Based on this, the LoGPC model [MF98] and the LogGPS model [IFH01] were proposed to incorporate network contention costs for irregular communication patterns and synchronization costs needed by MPI in sending long messages, respectively. The LoGPC model captures network contention using a variation of the M/G/1 queuing system. The LogGPS model captures the cost of extra REQ/ACK synchronization messages used by many MPI implementations to send very long messages.

In more recent distributed shared memory (DSM) machines, communication is usually off-loaded to a communication controller so that the processor is free to achieve greater concurrency. The parameter o in the LogP model no longer applies in this case. Holt, *et al.* [HHS+95] remedy this by changing the meaning of o to be the cost due to the occupancy of the communication controller, the time during which the controller is tied up handling a message and cannot process others.

The convergence of DSM machines to clusters of SMPs requires a new formulation of how the memory hierarchy affects the cost of message passing. While previous models are hardware-parameterized models, they fall short in this case because the effects of the memory hierarchy are mostly determined by the behavior of software that tries to exploit the hierarchy. Two software-parameterized models, the Memory logP model [CS03] and the $\log_n$P model [CG04], relate the cost of message passing with local memory reference patterns. In these two models, the cost of a message transfer is also a function of the distribution pattern of data that constitute the message. Although the nomenclature of the parameters is similar to that in the LogP model, some parameters have different meanings in the two models. The parameter o is now the cost of transferring an ideally distributed message, l represents the cost beyond o due to non-contiguous distribution, and g and P are the same as in LogP.

The $\log_n$P model is a generalization of the Memory LogP model. The memory system is generalized using the concept of middleware. Multiple layers of middleware are possible. The parameter n in this model stands for the number of implicit transfers along the data transfer path between two endpoints. For example, if the memory system is regarded as the only layer of middleware, then $n = 3$; corresponding to three implicit transfers, local memory to network interface, network transfer, and network interface to local memory. Both models assume $g = o$, thus the total cost of a message transfer is given by $w \times (o_1 + l_1 + o_2 + l_2 + o_3 + l_3)$.

6.1.2 A model for shared memory communication

Performance models based on the shared memory programming model are far less mature than the ones for message passing. The *check-in, check-out* (CICO) model [LCW93] is one worth mentioning. The CICO model is for cache-coherent shared memory computers. In this model communication arises only from cache misses and invalidation. The model annotates points in a program at which data moves in or out of cache. The communication cost is determined by the total amount of data transfered at these points.

The CICO model applies to shared memory programming on cache-coherent shared memory machines only. For shared memory programming on other architectures where the cache is not the only latency tolerance mechanism and/or the cache is noncoherent (e.g. UPC on DSMs), this model is not applicable because there might not be clear check-in and check-out points (for example, pipelined message transfers), and the communication cost is determined by multiple factors in addition to transfer sizes.

6.2 The UPC programming model

This section discusses some conspicuous features of the UPC programming model that complicate performance modeling. Communication in UPC is expressed *implicitly* as references to shared memory. This can be implemented in many ways, depending on the architecture. Although remote references are often implemented by messages, it is not realistic to model references using point-to-point communication because a reference may correspond to multiple messages and this correspondence varies from one implementation to another. The communication cost ultimately depends on both hardware-specific factors such as memory bandwidth, network bandwidth and latency, and program-specific factors such as reference patterns and data affinity.

The UPC programming model encourages *fine-grained accesses*. That is, references to scalar shared objects largely dominate communication. Given the increasing gap between local memory bandwidth and network bandwidth, it is reasonable to expect compilers and runtime systems to pursue aggressive latency avoidance and tolerance techniques. These techniques typically involve code transformations that change the number, order, and form of the shared references in the original program. Performance prediction based only on nominal access latency and the number of references leaves too many alternatives for explanation and is not accurate.

UPC has a memory consistency model in which references may be either *strict* or *relaxed*. Strict references are executed in program order and do not lend themselves to aggressive optimizations, but relaxed references offer many opportunities for optimizations. From a performance perspective, a practical UPC program should consist of a majority of relaxed references. Each compiler and runtime environment applies a different set of optimizations. Without detailed information about a particular implementation, it is difficult to model performance.

In summary, modeling UPC performance is challenging because its programming model is so far removed from the architectural model on which it is usually implemented. In addition, the connection between language constructs (e.g., remote references) and data movements (e.g., messages) is blurred by compiler and runtime optimizations. To tackle this problem, this work takes an approach based on platform benchmarking and dependence analysis. Platform benchmarking uses microbenchmarks to quantify a UPC platform's ability to perform certain common optimizations and dependence analysis determines which shared references in a code are potentially optimizable.

Above arguments about performance modeling challenges and solutions are also applicable to other PGAS languages, particularly Co-Array Fortran and Titanium. These languages and UPC share the same programming model, which determines that the performance of a program in most cases is reflected by the performance of shared memory accesses. Thus the same modeling approach that combines platform benchmarking and dependence analysis is suitable to these languages too.

6.3 Platform abstraction

The performance of a UPC program is determined by platform properties and application characteristics. A UPC platform can be characterized by two aspects: How fast the communication layer performs shared memory accesses and synchronization and how aggressively the UPC compiler and the runtime system optimize shared references. The baseline performance of a UPC platform, assuming no optimizations are done to shared memory accesses, can be characterized using parameters such as scalar access latency, shared access overhead, and average barrier cost. To characterize optimizations, internal knowledge about the particular UPC compiler and runtime system is needed but this is not always available. Analyses in this section show that a small set of optimizations that compilers and runtime systems might possibly perform can be abstracted. Then a set of corresponding microbenchmarks is used to test a particular platform's ability to perform these optimizations.

On the other hand, application parameters capture the effect of fine-grain reference patterns on the performance of a UPC computation. This work shows how to group shared memory accesses into categories that are amenable to the set of optimizations abstracted from studying UPC platforms. A UPC computation is characterized by the occurrences of each category. Combining this information with the platform measurements produces a performance prediction.

This section describes how to abstract UPC platform behaviors and the design of the microbenchmarks used to capture this abstraction. This section also discusses a dependence-based approach to parameterizing shared memory references in UPC applications.

Fine-grain access optimization is expected to be the focus of optimizations that a UPC platform conducts because fine-grain accesses tend to be the performance bottleneck [BHJ+03]. Although optimizations for private memory accesses commonly seen on sequential compilers are also expected to be performed by a UPC compiler, they are not considered in this paper due to their insignificance relative to the cost of remote accesses.

Latency avoidance and latency tolerance are the principal fine-grain access optimization techniques. The partitioned shared memory layout and the fine-grain access pattern determine that spatial locality and work overlapping are the two areas that UPC compilers and runtime systems will most likely exploit to achieve latency avoidance and latency tolerance. It is anticipated that the following optimizations may be performed by current UPC platforms. These optimizations either exploit spatial locality or work overlapping.

Access aggregation Multiple writes (puts) to shared memory locations that have affinity to a single UPC thread and are close to each other can be postponed and then combined into one put operation.

Multiple reads (gets) from locations that have affinity to the same UPC thread and are close to each other can also be coalesced by prefetching them into local temporaries using one get operation before they are used. For example, the two reads in the following code segment are subject to coalescing:

```
shared [] double *p;
  ... = *(p-1);
  ... = *(p+1);
```

Vectorization A special case of access aggregation is vectorized get and put operations for subscripted variables in a loop. Array accesses in a loop with fixed stride usually exhibit spatial locality that can be exploited using vector-like get and put operations. In the following simple example, both the read and the write make stride-1 accesses to remote locations. There are a total of N remote reads and N remote writes in the loop. If the reads and writes are performed using two vectors of length L, then the number of remote accesses can be reduced to N/L for both reads and writes. The vectorized code is shown on the right.

```
shared [] double *SA, *SB;
// SA and SB point to blocks
// of memory on remote threads.
for (i = 0; i < N; i++) {
  ... = SA[i];
  SB[i] = ...;
}
```

```
shared [] double *SA, *SB;
double tA[L], tB[L];
for (i = 0; i < N; i += L) {
  vector_get(tA, SA, i);
  for (ii = 0; ii < L; ii++) {
    ... = tA[ii];
    tB[ii] = ...; }
  vector_put(tB, SB, i); }
```

Remote access caching Accesses made to remote threads can be cached to exploit temporal and spatial reuse. This is a runtime optimization that can achieve effects similar to aggregation and vectorization but for shared memory accesses with regular patterns. Coalescing and vectorization, on the other hand, are compiler directed optimizations that, if properly implemented, can be effective for a wider range of access patterns.

Access pipelining Dependence-free accesses that appear consecutively in code can be pipelined to overlap with each other. Unlike the case of aggregation where multiple accesses are completed using one operation, here the number of operations does not change, so latency saved this way is limited. Accesses to locations with affinity to different threads can also be pipelined to overlap with each other, but this has the risk of jamming the communication network when the diversity of affinity increases.

Overlapping with computation Memory accesses can be issued as early as possible (for reads), or completed as late as possible (for writes), to hide the latency behind local computation. The potential benefits of this approach depend on both the freedom of moving read and write operations and the latency gap between shared memory access and private memory access.

Multi-streaming Remote accesses can be multi-streamed on platforms that support it. This is beneficial in situations where the memory system can handle multiple streams of data and there are a good number of independent accesses that can be evenly divided into different groups to be completed in parallel using multiple streams.

Note that the effects of these optimizations are not disjoint. For example, remote access caching can sometimes provide the effect of coalescing multiple accesses to the same remote thread. Vectorization can have an effect similar to caching accesses that exhibit spatial reuses. Pipelining and aggregation are both effective for independent accesses that appear in a sequence. In reality it is hard to tell exactly which optimizations lead to an observed effect on an application. But it is possible to use carefully designed microbenchmarks to determine which of these optimizations a particular UPC compiler and runtime system might have implemented.

6.4 Microbenchmarks design

Four microbenchmarks are proposed to capture a UPC platform's ability to optimize fine-grain shared memory accesses. They focus on capturing the effects of aggregation, vectorization and pipelining for remote accesses, as well as local shared access optimizations.

- **Baseline** This benchmark performs uniformly random read and write operations to the shared memory. Remote access caching (if available) is turned off. All UPC threads have the same workload. Accesses are made to remote shared memory. This benchmark obtains the latency of scalar remote accesses in an environment with a balanced communication pattern. Since random fine-grain accesses are generally not amenable to the optimizations listed in the previous section, this benchmark gives the baseline performance of a UPC platform.

- **Vector** Each UPC thread accesses consecutive memory locations, a vector, starting from a random location in a large shared array with indefinite block size. This benchmark determines if the platform exploits spatial locality by issuing vectorized reads and writes or by caching remote accesses.

- **Coalesce** In this case each UPC thread makes a sequence of accesses with irregular but small strides to a large shared array with indefinite block size. The accesses appear as a sequence in the microbenchmark code to allow pipelining, but if access aggregation is supported by the platform, this access pattern is more amenable to aggregation. This benchmark captures the effect of access pipelining and aggregation.

- **Local *vs.* private** Each UPC thread accesses random memory locations with which it has affinity. Then the same operations are performed to private memory locations. The cost difference between the two kinds of accesses represents shared access overhead, i.e., the software overhead arising from processing shared memory addresses.

These microbenchmarks represent four typical reference patterns found in UPC programs. Each pattern results in an effective memory access rate in terms of *double words per second*. Let $S_{baseline}$, S_{vector}, $S_{coalesce}$ and S_{local} be the rates of the four patterns.

The measured data access rates for these four patterns on some UPC platforms are listed in Tables 6.1–6.4.

6.5 Application analysis

The whole purpose of UPC compiler and runtime optimizations is to exploit concurrency so that multiple shared memory operations can be scheduled in parallel. The Bernstein conditions [Ber66]

	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC	
THREADS	read	write	read	write	read	write	read	write
2	71.4	28.1	43.0	87.2	47.6	21.5	2.2	0.77
4	76.8	28.0	72.3	114.5	61.5	22.0	2.3	0.76
6	80.0	30.8	83.2	119.4	63.0	22.2	2.5	0.92
8	79.6	30.0	88.0	126.9	63.8	22.0	2.2	0.77
10	80.0	29.0	90.4	135.2	65.5	23.1	2.2	0.77
12	83.1	32.5	92.8	137.7	66.0	23.1	2.5	0.93

Table 6.1: Baseline pattern measurements (microseconds/double word)

	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC	
THREADS	read	write	read	write	read	write	read	write
2	60.8	22.0	1.0	1.0	47.4	21.2	2.0	0.6
4	67.8	21.9	0.8	1.3	64.2	21.8	2.1	0.6
6	68.2	22.8	0.8	1.3	65.9	21.8	2.2	0.6
8	69.5	22.2	0.8	1.2	66.2	21.8	2.1	0.6
10	69.4	22.0	1.0	1.0	67.7	22.0	2.2	0.6
12	71.2	28.9	1.3	1.4	68.5	22.3	2.2	0.6

Table 6.2: Vector pattern measurements (microseconds/double word)

established the constraints for concurrent scheduling of memory operations. That is, dependence-free accesses can be safely executed in parallel. Following [AK02], dependence is defined as: *A dependence exists between two memory references if (1) both references access the same memory location and at least one reference stores to it, and (2) there is a feasible execution path from one reference to another.* Based on this definition, dependences can be categorized as *true dependence*, *antidependence*, and *output dependence*. In this study, the concept of *input dependence* is also used, i.e., both references involved in a dependence are reads.

The UPC memory model forces another constraint for concurrent scheduling of memory accesses. It prohibits reordering strict operations and reordering strict and relaxed operations. References separated by strict operations must complete in program order even if they are independent of each other. Strict operations, including strict memory accesses, barriers, fences, and library function calls such as collectives, are defined as *sequence points*. Effectively, there is a true dependence from every statement before a sequence point to the sequence point, and a true dependence from it to every statement after it. In other words, sequence points divide a program into a series of

	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC	
THREADS	read	write	read	write	read	write	read	write
2	60.0	22.8	12.2	14.3	5.8	21.3	2.0	0.64
4	74.3	25.8	12.2	14.6	7.7	21.7	2.1	0.64
6	69.2	22.0	13.1	15.8	7.8	21.9	2.2	0.64
8	70.3	21.8	14.4	18.4	7.9	22.0	2.2	0.65
10	75.2	22.6	17.4	20.2	8.2	22.3	2.2	0.64
12	77.3	25.3	15.6	21.6	8.2	22.5	2.5	0.65

Table 6.3: Coalesce pattern measurements (microseconds/double word)

THREADS	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC	
	read	write	read	write	read	write	read	write
2	0.12	0.12	0.12	0.12	0.12	0.15	0.8	1.44
4	0.12	0.12	0.12	0.12	0.12	0.15	0.8	1.44
6	0.12	0.12	0.12	0.12	0.12	0.16	1.0	1.6
8	0.12	0.12	0.12	0.12	0.12	0.15	0.8	1.44
10	0.12	0.12	0.12	0.12	0.15	0.2	1.0	1.6
12	0.12	0.12	0.12	0.12	0.15	0.2	1.0	1.6

Table 6.4: Local pattern measurements (microseconds/double word)

intervals.

A dependence-based analysis of a UPC program can identify candidate references for concurrent scheduling in an interval. First, a dependence graph is constructed for all references inside an interval. Then, references to a shared array are partitioned into groups based on the four reference patterns represented by the microbenchmarks described in section 6.4, under the assumption that their accesses are amenable to the optimizations targeted by these patterns. References in the same group are subject to concurrent scheduling. Last, their collective effects are aggregated to predict the performance of the program.

To precisely describe reference partitioning, it is necessary to formally define a partition as a 3-tuple $(C, pattern, name)$, where C is the set of references grouped in the partition. *pattern* is one of the four patterns, *baseline*, *vector*, *coalesce*, or *local*, and some simple combinations of them. Simple combinations are allowed because accesses caused by a reference may incur different costs at different time during an execution. For example, some accesses are local and others are remote. *name* is the name of the shared object referenced by C, which implies that all references in a partition access the same shared object because references to different shared objects are not amenable to aggregation.

Mathematically, a partition is an *equivalence class* with the *equivalence relation* being "same pattern". Every reference must be in one partition and can only be in that partition.

The following sections discuss the principles of reference partitioning. To facilitate the analysis, user defined functions are inlined to get a flat code structure. Recursive routines are not considered in this study.

6.5.1 Reference partitioning

Reference partitioning can be reduced to a variation of the *typed fusion* problem [AK02]. Thus, the *reference partitioning* problem is formulated as the following.

Let $G = (V, E)$ be a dependence graph of an interval, where V is a set of vertices denoting shared references appearing in the interval (each vertex denotes a separate reference), and E is a set of edges denoting dependences among the references. Let T be the set of *names* that label the vertices in V. A *name* uniquely identifies the shared object involved in the reference. Alias analysis must be done so that aliases to the same object have the same name. Let $B \subseteq E$ be a set of edges denoting true dependences and antidependences. Then the reference partitioning graph $G' = (V', E')$ is the graph derived from G that has a minimum number of edges by grouping vertices in V. Vertices are grouped subject to the following constraints:

1. Vertices in a partition must have the same name t, where $t \in T$.

2. At any time, the memory locations referenced by vertices in a partition must have the same affinity.

3. No two vertices joined by an edge $e \in B$ may be in the same partition.

In the resulting graph G', each vertex may contain more than one reference. If multiple references in a partition access the same memory location, then they should be counted only once because only one access is really needed. Each reference in a partition incurs a uniform cost determined by the pattern of the partition. A partition has a cost that is just the aggregated costs of its members.

What pattern a partition should assume, and consequently its cost, is determined by what optimizations are applicable to the references in the partition on a particular UPC platform. Consider the following two examples:

```
shared [] float *A;
// A points to a block of
// memory on a remote thread
for (i = 1; i < N; i++)
{   ... = A[i];
    ... = A[i-1]; }
```

```
shared float *B;
for (i = 1; i < N; i++)
{   ... = B[i];
    ... = B[i-1]; }
```

In the example on the left, references to `A[i]` and `A[i-1]` are in one partition. If the platform supports access vectorization then they are vectorizable because all accesses have the same affinity and are unit-strided. This partition is assigned the *vector* pattern. On the other hand, if the platform does not support access vectorization but supports coalescing then the two references can be coalesced into one access on each iteration and the partition is assigned the *coalesce* pattern. Finally, if the platform does not support vectorization or coalescing then the partition is assigned the *baseline* pattern.

In example on the right, the two references to `B` appear to be similar to the two references to `A` in the previous example. But `B[i]` and `B[i-1]` are in two separate partitions because they access locations with different affinities on each iteration. Neither of the two partitions are subject to vectorization because they both access locations with different affinities across iterations. The two partitions can only be assigned a mixed *baseline-local* pattern, because for every `THREADS` accesses there is one *local*, no matter what optimizations a platform supports. For example, if `THREADS` $= 4$ then it will be a (75% *baseline*, 25% *local*) pattern.

6.6 Performance prediction

Each reference partition within an interval identified from dependence analysis is associated with a cost that expresses the number of accesses in the group and the access pattern of the group. The communication cost of the interval is modeled by summing the costs of all reference partitions. Specifically, the following equation defines the communication cost for any interval i to be the sum of the costs over all reference groups in that interval:

$$T^i_{comm} = \sum_{j=1}^{Groups} \left(\frac{N_j}{r(N_j, pattern)} \right) \tag{6.1}$$

where N_j is the number of shared memory accesses in any reference group j and $r(N_j, pattern)$ gives the effective data transfer rate (double words per second) of the pattern associated with the group, which is a function of the number of accesses and the pattern of accesses. The values of $r(N_i, pattern)$ are obtained by benchmarking the four patterns on a UPC platform with varying numbers of accesses.

On the other hand, the computation cost of an interval T_{comp} can be modeled by simulating the computation using only private memory accesses. The run time of an interval is simply predicted to be $T_{comm} + T_{comp}$. The run time of a thread is the sum of the costs of all intervals, plus the costs of barriers, whose costs are also estimated by benchmarking. When it is necessary to take control flow into consideration, only the intervals on the critical path of execution are considered. Finally, the thread with the highest predicted cost is taken to be the cost of the whole program.

The *degree of concurrency* supported by a platform is determined by the gap between the speed of private memory access and the speed of shared memory access. Let $S_{private}$ be the private memory access speed, which can be obtained using an existing microbenchmark such as STREAM [McC06], and let $S_{baseline}$, S_{vector}, $S_{coalesce}$, and S_{local} be as defined as in Section 6.4. Then, $G_{baseline}$, G_{vector}, $G_{coalesce}$ and G_{local} are the degrees of concurrency with respect to the 4 canonical shared memory access patterns, respectively:

$$G_{baseline} = \frac{S_{baseline}}{S_{private}}, \quad G_{vector} = \frac{S_{vector}}{S_{private}}, \quad G_{coalesce} = \frac{S_{coalesce}}{S_{private}}, \quad G_{local} = \frac{S_{local}}{S_{private}}$$

These values give an upper bound on how much communication can overlap with computation. Smaller values mean that the shared memory access speed is relatively faster and a shared memory access can overlap with less local computation. For example, if a G value is 4 then a shared access can overlap at most 2 floating point operations (each with 2 operands).

Let N_s be the number of memory accesses in the sequential code. Let N_c, N_l, and N_r be the average number of private memory accesses, local shared memory accesses, and remote shared memory accesses issued by any thread in the parallelized code, respectively. Let N_p be the normalized average number of memory accesses issued by any thread in the parallelized code. That is, $N_p = N_c + (N_l \times G_{local}) + (N_r \times G_{remote})$, where G_{remote} is the weighted average of $G_{baseline}$, G_{vector}, and $G_{coalesce}$. Then the speedup achievable by the parallelized code is given by the ratio N_s/N_p:

$$S = \frac{T_s}{T_p} \approx \frac{N_s}{N_p} = \frac{N_s}{N_c + (N_l \times G_{local} + N_r \times G_{remote})} \tag{6.2}$$

where T_s is the sequential run time and T_p is the parallel run time prediction based on Equation 6.1. Oftentimes, private memory accesses are orders of magnitude faster than shared memory accesses because they are direct memory loads and stores instead of being implemented using runtime function calls, so N_c is negligible unless it is of the same or larger order than than $(N_l + N_r)$. Note that this motivates an important optimization desired by UPC applications: the privatization of local shared memory accesses. When a compiler fails to do this, it is always beneficial for a programmer to manually cast pointers to local shared memory locations into pointers to private (i.e., regular C pointers), whenever possible.

6.7 Performance modeling for Co-Array Fortran and Titanium

Extending UPC performance modeling techniques to other PGAS languages is possible but not trivial. This section discusses how this may be done for Co-Array Fortran and Titanium.

Both languages support some abstractions not supported by UPC. For example, Co-Array Fortran has a rich set of multi-dimensional array operations. Titanium supports classes, templates, operator overloading, exceptions, etc. These features improve programmability and maintainability for application development in their respective domains. But when it comes to performance the most important factors are interprocess communication and synchronization, which are expressed using the same abstractions in these languages. It is this common ground that makes it possible to extend UPC performance modeling approach to the other two languages.

As described earlier in this chapter, the performance of a UPC program is determined by two things, the fitness of the platform on which the program runs and the shared memory reference patterns of the program. The former is measured by platform benchmarking and the latter is characterized by static analysis. It is straightforward to translate the UPC microbenchmarks described in Section 6.4 into Co-Array Fortran and Titanium to create equivalent platform abstraction tools. But the static analysis procedures must be different due to the vast syntactical difference between these languages. The principles of static analysis, however, are still the same: identifying shared memory references and partitioning them into equivalence classes. The remaining part of this section outlines issues that need to be addressed in static analysis for Co-Array Fortran and Titanium.

6.7.1 Co-Array Fortran

Static analysis for Co-Array Fortran is generally easier than for UPC. This is mainly because of a salient difference between the shared array indexing schemes used in Co-Array Fortran and UPC: A co-array is indexed using "co-dimensions" that include a *image (process) index*. Therefore, whether a reference is to local data or remote data is always obvious.

Co-arrays are the only type of shared objects in Co-Array Fortran, they can only be statically allocated. There is no block size in the type system of co-arrays (or in equivalent UPC jargon, all co-arrays have indefinite block size). Moreover, indirect references such as pointers and aliases are not widely used. These factors also contribute to the easiness of static analysis of Co-Array Fortran code.

Synchronization operations in Co-Array Fortran can find counterparts in UPC. `SYNC_ALL` corresponds to `upc_barrier`; `SYNC_MEMORY` corresponds to `upc_fence`; critical sections correspond to UPC lock and unlock operations. Just as in UPC, these synchronization operations divide a piece of Co-Array Fortran into intervals.

As an extension to Fortran 95, Co-Array Fortran is powerful in expressing multi-dimensional arrays and their operations. A whole array or any slice of an array can be used as a unit in an operation, which means a single statement may contain multiple references to the elements in multiple co-arrays. This is the only complexity that may arise from Co-Array Fortran analysis, but an automated static analysis tool, for example, the middle-end of a Co-Array Fortran compiler, can handle this very well.

6.7.2 Titanium

Titanium is the most difficult one among the three languages. Titanium is a language designed to target irregular problems such as adaptive mesh refinement (AMR). It supports highly flexible global address space manipulation that makes it easy to construct and demolish distributed data structures, but difficult to track the accesses to these data structures. The main reasons that make it difficult include:

- There are no statically declared shared objects (a sharp contrast to Co-Array Fortran). Titanium's global address space is totally based on dynamic allocation. Each process allocates its own "sub-grids" that are accessible by other processes. Sizes and shapes of the sub-grids across processes may be different.
- Shared objects are usually accessed by pointer dereferencing, not by indexing.

In addition, the high-level abstraction power inherited from Java also makes it difficult to analyze shared memory accesses. An access may be buried many levels down to the abstraction hierarchy.

Nonetheless, Titanium does provide several hints that make static analysis still tractable. The *local* and *global* quantifiers specify whether a pointer is used to dereference private objects or shared objects. Interprocess communication in Titanium is achieved in two ways: By dereferencing a global pointer to a remote object or by calling communication functions such as `Exchange` and `Broadcast`. Consequently, as long as the static analysis can carefully track the dereferencing of all global pointers then the dependence analysis and reference partitioning can be performed as they are done in UPC. `Exchange` and `Broadcast` are similar to the corresponding UPC collectives. Accounting for their cost is relatively easy because they are explicit communication functions.

Based on above understanding, the static code analysis techniques used for UPC is still applicable to Titanium but not enough. These techniques must be enhanced to be able to perform thorough indirection analysis. Manual analysis is unrealistic, automated solution is needed.

6.8 Summary

On today's parallel platforms, communication cost still dominates application performance. In the case of UPC, communication cost is represented by a "shared memory wall", the gap between shared memory bandwidth and processor bandwidth. Therefore, studying shared memory access patterns is a reasonable way of understanding UPC performance. However, UPC programs are inherently difficult to model for two reasons. First, the UPC programming model entails implicit, finei grain shared memory accesses that lead to sophisticated communication patterns in a dynamic execution environment. Second, the fine grain shared memory accesses are subject to UPC compiler optimizations that further obscure access pattern analysis.

This study is the first performance modeling effort for UPC programs. It proposes an approach to analyze the implicit, finegrain shared memory accesses in UPC programs. The approach recognizes four basic reference patterns and accordingly uses four simple microbenchmarks to measure a UPC platform's ability to optimize fine grain shared memory accesses. Next, a dependence-based analysis is used to partition references in an application into groups and to associate each group with a certain pattern or a simple combination of patterns. The cost of each group is determined by the pattern associated with the group and the number of shared memory accesses made by the group. The run time of an application is determined by the aggregated costs of all reference groups.

As shown in the next chapter, models built with this approach predict the run times of three applications running on three different UPC platforms. These predictions have a maximum error of $\pm 15\%$ in most cases. This is good accuracy for an analytical performance model. Although the validation is done only with respect to UPC, it is expected that performance models for Co-Array Fortran and Titanium can be constructed in a similar way. The key is to identify and correctly characterize shared memory references through static analysis, a task that may need sophisticated software tools in the case of Titanium because of many high level abstractions imposed by the language.

Chapter 7

Performance model validation

This chapter provides validation for the performance modeling method presented in the previous chapter. Performance modeling is an important tool for application developers to understand an application's performance behavior, as well as for system engineers to evaluate a parallel system and help them make design decisions. To demonstrate the usefulness of the performance modeling method in this two aspects, this chapter first builds basic models to predict the run time of three simple UPC application kernels and validate the prediction with actual measurements. Then the models are extended to give better explanations for the effects of remote reference caching in the MuPC runtime system.

7.1 Modeling the performance of UPC application kernels

The three application kernels include a histogramming program, naïve matrix multiply, and Sobel edge detection. The three applications feature three memory access patterns encountered in many real-world applications. The histogramming code contains a large number of random, fine-grain shared memory updates and is communication intensive. The matrix multiply code accesses memory in a regular pattern and all remote accesses are reads. This program is also communication intensive. The Sobel edge detection code is computation intensive and most accesses are made to local shared memory.

Performance models are built to predict the run times of the three programs running on three different UPC platforms with fixed problem sizes. The prediction is validated by comparing the actual run time and the predicted run time. The precision of prediction is defined to be:

$$\delta = \frac{predicted\ cost - actual\ cost}{actual\ cost} \times 100\% \tag{7.1}$$

The experiments were done using MuPC V1.1.2 beta [ZSS06, ZS05, Mic05], Berkeley UPC V2.2 [UC 04a, CBD+03], and GCC UPC V3.4.4 compilers [Int04]. MuPC and Berkeley UPC are run on a 16-node Intel 2.0 GHz x86 Linux cluster with a Myrinet interconnect. GCC UPC is run on a 48-PE 300MHz Cray T3E.

MuPC and Berkeley UPC take a similar approach in providing a compilation and execution environment for UPC programs. UPC code is first translated into C code with UPC constructs being replaced with corresponding C constructs and runtime function calls. The translated C code is then compiled using a regular C compiler and linked to a runtime library to produce an executable. The

MuPC runtime incorporates a software cache for remote shared memory accesses as a latency tolerance mechanism. Berkeley UPC provides some source-level optimizations as experimental features, including more efficient local shared pointer arithmetic, remote access coalescing, communication-computation overlapping, and redundancy elimination for share address computation. On the other hand, GCC UPC directly extends the GNU GCC compiler by implementing UPC as a C language dialect. GCC UPC currently provides no optimizations beyond sequential code optimizations.

7.1.1 Histogramming

In this application a cyclically distributed shared array serves as a histogramming table. Each UPC thread repeatedly chooses a random element and increments it as shown in the following code segment:

```
shared int array[N];
for (i = 0; i < N*percentage; i++) {
  array[loc]++;   // loc is a pre-computed random number
}
upc_barrier;
```

The parameter `percentage` determines how many trips the loop iterates, thus how big a portion of the table will be updated. In this simple setting, collisions obviously occur. The probability of collisions increases as `percentage` increases. This example models the run times while varying `percentage` from 10% to 90% with a 10% interval.

This code cannot be optimized by coalescing or vectorization because the table element accessed in each step is randomly chosen. Assume the elements chosen by each thread are uniformly and randomly distributed, then $1/\texttt{THREADS}$ of the updates are made to locations within a thread's affinity and $(\texttt{THREADS}-1)/\texttt{THREADS}$ of the updates are made to remote locations. The only reference partition contains only `array[loc]`, which fits a mixed *baseline-local* pattern. Run time prediction is thus based on the number of local shared accesses, the number of remote accesses, and the effective data transfer rates obtained using the *baseline* and the *local* microbenchmarks. The results in Table 7.1 show that the model very accurately predicted the run time for all three platforms. The largest relative error is less than 10% and in most cases it is less than 5%.

7.1.2 Matrix multiply

The matrix multiply program is a naïve version of the $O(N^3)$ sequential algorithm. The product of two square matrices ($C = A \times B$) is computed as follows [EGC01]:

```
upc_forall(i=0; i<N; i++; &A[i][0]) {
  for(j=0; j<N; j++) {
    C[i][j] = 0;
    for (k=0; k<N; k++)
      C[i][j] += A[i][k]*B[k][j]; }
}
upc_barrier;
```

Percentage	δ (%)		
	MuPC	Berkeley UPC	GCC UPC
10%	-2.2	-0.38	-3.6
20%	-4.8	-0.25	-3.5
30%	-0.4	0.15	-3.5
40%	-4.0	0.32	-3.5
50%	1.6	0.45	-3.5
60%	-9.8	0.36	-3.6
70%	-4.6	0.39	-3.5
80%	-3.3	0.35	-3.5
90%	-9.7	0.54	-3.5

Table 7.1: Prediction precision for histogramming. The size of the histogram table is $1M$, THREADS $= 12$. The results are averages of at least 10 test runs.

To facilitate this computation the rows of A and C are distributed across threads, while columns of B are distributed across threads. Both row distribution and column distribution can be either *cyclic striped* or *block striped*, that is, matrices are declared in either of the following two ways (in the experiments N is always divisible by THREADS):

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

cyclic striped

```
#define M1 (N*N/THREADS)
#define M2 (N/THREADS)
shared [M1] double A[N][N];
shared [M2] double B[N][N];
shared [M1] double C[N][N];
```

block striped

Memory access patterns are similar in both distributions. Accesses to A are all local reads. The majority of accesses to B are remote reads, with a portion being local reads. Accesses to C involve both local reads and local writes. The numbers of all types of accesses made by each thread can be easily counted, and these numbers are the same for both distribution schemes.

Reference partitioning identifies the following partitions: (1) A partition containing a reference writing to C[i][j]. There are two references of this type but they always access the same location in each iteration of the j-loop, so they are included only once. This partition fits the *local* pattern. (2) A partition containing a reference reading from C[i][j] as implied by the $+=$ operation. This partition also fits the *local* pattern. (3) A partition containing a reference reading from A[i][j] that fits the *local* pattern. (4) A partition containing a reference to B[k][j] that fits a mixed *vector-local* pattern.

This analysis shows that a majority of accesses by the references in partition (4) are subject to remote access vectorization. Currently, none of the three UPC implementations detect this opportunity for optimization, but remote access caching by MuPC can achieve effects similar to vectorization in this case because spatial locality can be exploited.

The modeling results are shown in Table 7.2. The negative errors in the cases of Berkeley UPC and GCC UPC show an underestimation of run times for these two platforms. It is suspected that there

THREADS	δ (%)							
	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC UPC[1]	
	cyclic striped	block striped	cyclic striped	block striped	cyclic striped	block striped	cyclic striped	block striped
2	-12.2	0.7	7.9	-2.1	-1.2	-1.9	-4.0	-14.0
4	-4.5	0.3	15.3	19.5	-7.4	-14.7	-8.8	-10.5
6	3.6	-0.7	15.8	11.8	-4.5	-8.9	7.6	-2.0
8	2.2	-4.4	9.8	13.0	-7.0	-12.0	10.6	9.6
10	-0.4	-2.2	3.9	2.0	-7.4	-15.2	6.2	-3.2
12	2.9	0.9	9.8	8.8	-5.4	4.4	3.1	-5.7

Table 7.2: Prediction precision for matrix multiply. The size of matrices are 240×240 (doubles). The results are averages of at least 10 test runs.

are some non-UPC related factors that lead to increased costs. The relatively larger error for MuPC with cache when running with more than two threads represents an overestimation of run times. This is because the cache also exploited temporal locality (i.e., many cache lines are reused) and led to extra savings. However, the model did not capture this because the model regarded the cache only as a simulated vectorization mechanism.

7.1.3 Sobel edge detection

In this classical image transforming algorithm, each pixel is computed using information of its direct neighbors. An image of size $N \times N$ is distributed across threads so that each thread has $N/$`THREADS` contiguous rows. Communication is needed for the border rows only. Local shared memory references are the predominant performance factor. The kernel of this code [EGC01] is shown below.

```
#define B (N*N/THREADS)
shared [B] int O[N][N], E[N][N];
upc_forall(i=1; i<N-1; i++; &E[i][0]){
  for (j=1; j<N-1; j++) {
    d1 =   O[i-1][j+1] - O[i-1][j-1];
    d1 += (O[i][j+1]   - O[i][j-1])<<1;
    d1 +=  O[i+1][j+1] - O[i+1][j-1];

    d2 =   O[i-1][j-1] - O[i+1][j-1];
    d2 += (O[i-1][j]   - O[i+1][j])<<1;
    d2 +=  O[i-1][j+1] - O[i+1][j+1];

    m = sqrt((double)(d1*d1+d2*d2));
    E[i][j] = m > 255 ? 255 : (unsigned char)m; }
}
```

Reference partitioning results in the following partitions: (1) A partition containing the reference `E[i][j]` that involves only local shared writes. This partition fits the *local* pattern. (2) A partition

[1]GCC UPC array references with two subscripts are 20 – 60% more expensive than array references with one subscript. It is believed to be a performance bug. Numbers shown in the table take this into consideration.

containing references to the `i`-th row of `0`. This partition also fits the *local* pattern because all accesses are local shared reads. (3) A partition containing references to the `(i-1)`-th row of `0`. (4) A partition containing references to the `(i+1)`-th row of `0`. Partitions (3) and (4) fit a mixed *local-vector* pattern on the MuPC platform due to the exploitation of spatial locality by MuPC's cache, but they fit a mixed *local-coalesce* pattern on the Berkeley UPC platform because the coalescing optimization enabled by Berkeley's compiler is applicable. They fit a mixed *local-baseline* pattern on the GCC UPC platform because no optimizations are performed by this platform.

The modeling results are shown in Table 7.3. A large error (-21.6%) occurs in the case of MuPC with cache enabled when running with two threads. This implies unaccounted for cache overhead. With only two threads the communication is minimal and the benefit of caching is not big enough to offset the overhead. Again, simulating access vectorization using a remote reference cache partially accounts for other errors in the case of MuPC with cache.

	δ (%)			
THREADS	MuPC w/o cache	MuPC w/ cache	Berkeley UPC	GCC UPC [1]
2	4.8	-21.6	7.3	-10.3
4	1.4	15.8	8.3	-10.8
6	11.8	16.3	1.1	-6.3
8	9.9	16.8	-1.3	7.5
10	7.0	17.5	-4.3	-1.0
12	-0.5	14.3	5.0	-3.5

Table 7.3: Prediction precision for Sobel edge detection. The image size is 2000×2000 (integers). The results are averages of at least 10 test runs.

7.2 Caching and performance prediction

In the previous section, prediction errors are relatively larger for cases where remote reference caching is used. This section tries to remedy this by including cache behavior into the performance model.

To simplify the discussion, the analysis focuses only on shared reads. All caches are effective for reads, but not all are effective for writes. For example, the cache in HP-UPC does not cache shared writes at all. On the other hand, the principles of caching for shared reads can be easily extended to shared writes by considering a few extra complexities such as the policy for invalidation (write-through or write-back) and the cost and frequency of invalidations.

Equation 6.1 predicts the communication cost of a computation interval. It takes as input the number of shared memory accesses of different reference patterns inside the interval and the data transfer rates of corresponding patterns. Data transfer rates are determined by UPC platform characteristics. The variety of reference patterns and the number of accesses of each pattern are determined by application characteristics. Although remote reference caching is a platform-level feature, it does not affect data transfer rates, which are fixed by the platform's underlying communication hardware and library. But remote reference caching effectively alters reference patterns. It turns some remote shared accesses (cache hits) into local shared accesses at the cost of a miss penalty. Therefore, the following two modifications to equation 6.1 are needed:

- The number of cache hits is subtracted from the number of remote accesses and added to the number of local shared accesses.

- For the remaining remote accesses, each is a cache miss and causes a block to be brought into cache, so the value of remote access latency is increased to equal the miss penalty. A cache miss is handled by copying consecutive bytes equivalent to the size of a cache block from the shared memory to the cache using the `upc_memcpy()` function. The cost of this copying is the miss penalty and can be measured by benchmarking.

Among the four basic reference patterns identified in chapter 6, *baseline*, *vector*, and *coalesce* are about remote accesses and will be affected by above modifications. The *baseline* accesses are cache unfriendly and their quantity is unlikely to change. The *coalesce* accesses exhibit some temporal locality and may benefit from the cache. The *vector* accesses are all cache friendly and most of them turn into local shared accesses. A careful execution trace analysis will help show how many accesses of each of the three types actually become local accesses. For programs with simple reference patterns, however, run time cache statistics will reveal the same information. Both MuPC and HP-UPC can generate cache statistics such as the number of cache misses/hits. As an example, this information is used to reconcile the actual run time and predicted run time for the matrix multiply (cyclic striped distribution) modeled in chapter 6. The results are plotted in figure 7.1. The figure shows an overall improvement for prediction precision, compared to the predictions in Chapter 6.

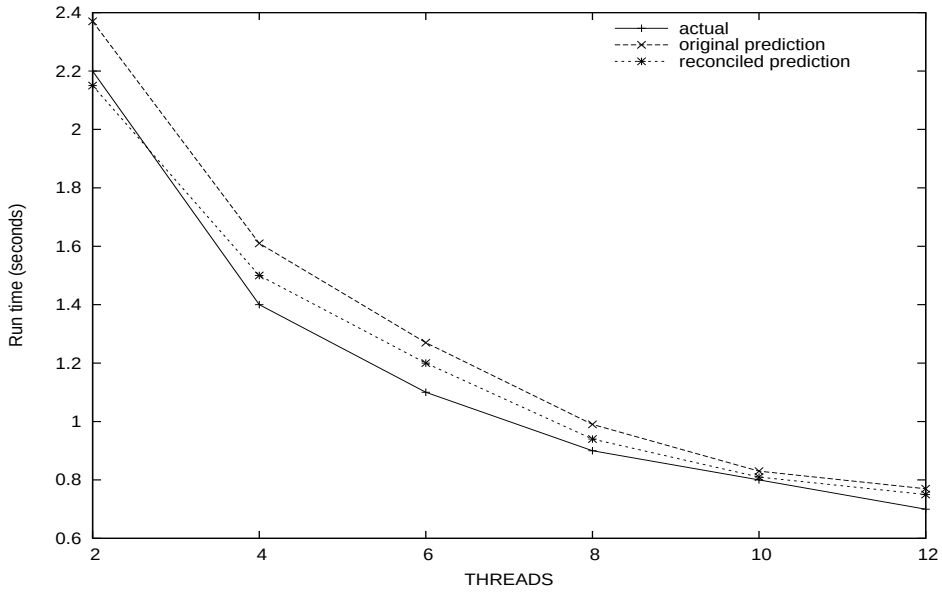

Figure 7.1: Cost prediction reconciled using MuPC cache statistics for matrix multiply (cyclic striped distribution)

Chapter 8

Applications of the UPC performance model

This chapter is a recapitulation of findings of the performance modeling work presented in this report, and observations of what can be extended from the current research. Discussions here focus on UPC because they are based on UPC platform benchmarking results and UPC language particularities. The discussions cover the applications of UPC performance modeling in the following areas:

- Characterizing UPC platform and identifying performance bottlenecks
- Revealing application performance characteristics
- Recognizing favorable programming styles with respect to a UPC platform

Then the last part of this chapter envisions future UPC work based on the current findings.

8.1 Characterizing selected UPC platforms

Many benchmarks are used throughout this research. Some of them are established popular benchmarks for parallel systems, for example, the UPC version of the NPB suite. These types of benchmarks are useful at the early stage to demonstrate the usefulness of a UPC platform, but they have been long regarded as not appropriate for UPC because results produced by these benchmarks cannot be easily mapped to UPC language features. An important contribution of this research is that it has created a set of microbenchmarks that reliably captures a UPC platform's characteristics. These microbenchmarks define a set of elementary fine-grain memory operations that are pervasively used in UPC programs and whose performance can be easily quantified. So a UPC platform can be characterized based on its ability to perform these operations. This approach provides simple, straightforward, yet meaningful criteria to compare UPC platforms.

Table 8.1 lists these microbenchmarks and the basic operations they measure respectively.

For any UPC platform, this set of microbenchmarks produces a performance matrix that is centered around shared memory operations. The performance matrix dictates that a good UPC platform must be aware of access patterns, that is, it not only provides low latencies for remote random accesses, but also differentiates between randomness and non-randomness, and between remote and local. Given the performance matrix, an ideal UPC platform can be identified using the following criteria:

[1]The Apex-MAP used in this research is adapted from the original version described in [SS05].

Microbenchmarks	*Operations measured*	*Bottlenecks exposed*
stream	Remote accesses issued by a single thread with others being idle	Nominal random access latency (the lower bound of random access latency)
baseline	Remote accesses issued from all threads	Achievable random access latency
coalesce	Remote accesses with clustered but random pattern	Capability of access aggregation or pipelining
vector	Vectorizable remote accesses	Capability of Access vectorization, remote reference caching for spatial locality
local	Accesses to the shared memory with local affinity	Random local access latency
Apex-MAP[1]	Remote accesses with tunable temporal locality and spatial locality	Effectiveness of remote reference caching

Table 8.1: UPC platform characterization microbenchmarks

- The *stream* and *baseline* latencies being on the same scale.
- The *local* latencies being on the same scale with regular private random accesses, and at least $1 \sim 2$ orders of magnitude lower than *stream* and *baseline*.
- The *coalesce* and *vector* latencies being on the same scale with *local*.
- The bandwidths for *ring* and *flush* being on the same scale and both scaling with the number of threads.

Microbenchmarks	*MuPC w/o cache*	*MuPC w/ cache*	*HP-UPC w/o cache*	*HP-UPC w/ cache*	*Berkeley UPC*
stream	20.0	25.4	8.2	8.4	13.6
baseline	83.1	92.8	10.4	9.8	66.0
coalesce	77.3	15.6	9.9	2.8	8.2
vector	71.2	1.3	10.1	2.4	68.5
local	0.12	0.12	0.09	0.09	0.15

Table 8.2: Performance matrices for selected UPC platforms (THREADS = 12)

As an example, Table 8.2 shows the latencies in microseconds obtained from running these microbenchmarks on MuPC, HP-UPC, and Berkeley UPC. The performance matrices clearly expose performance bottlenecks for these platforms. When evaluating the numbers against the criteria, none of these platforms is an ideal UPC platform:

- HP-UPC with cache is the most optimized platform. But the latencies of *coalesce* and *vector* are still one order of magnitude bigger than the *local* latency.
- MuPC with cache has similar problems. Besides, its *baseline* latency is 4 times the *stream* latency.

- Berkeley UPC is only able to exploit the *coalesce* pattern. And this is only available as an experimental feature.
- Platforms without cache hardly meet any of the above criteria.
- All platforms exhibit similar *local* latency, which is about one order of magnitude bigger than that of private memory accesses (The typical latency of private memory accesses is on the par of 10 nanoseconds).
- All platforms suffer seriously from shared memory bank contentions. The *flush* latency is $2 \sim 5$ times the *ring* latency.

8.2 Revealing application performance characteristics

Another important contribution of this work is the UPC performance model that makes use of the performance matrix and the accompanying theory of breaking a UPC program down into simple work units based on reference pattern analysis. These work units can be directly mapped to the types of access patterns characterized by the microbenchmarks. So the performance of the UPC program can be explained by combining the amount of the work units with the corresponding values in a UPC platform's performance matrix. Examples of the applications of the UPC performance modeling in this area include performance prediction. Chapter 7 has shown that application run times predicted via performance modeling matched actual run times well.

There are times when quantitative performance prediction is unnecessary or not easy to obtain. For example, it may take a long time to model a real-world application with complicated structure. Often, a user only wants to know which platform is a more suitable one for this application. As long as there are some qualitative knowledge about the memory access pattern of the application, the user can choose a good platform for this application based on the information presented in Table 8.2. Taking the CG program in the NPB benchmark suite as an example, the performance of this program is illustrated in Figure 4.6 (Chapter 4). CG is an implementation of the conjugated gradient method for unstructured sparse matrices. Access patterns in CG are complicated but the *vector* pattern predominates. Based on Table 8.2, HP-UPC should be the best platform for this program, followed by MuPC with caching, then followed by Berkeley UPC. Figure 4.6 confirms this prediction.

8.3 Choosing favorable programming styles

Finally, the performance model is more of a valuable tool in guiding program design. The UPC language offers an affinity-aware shared memory programming model that features fine-grain remote accesses. But it also provides constructs for programming in a coarse-grain message passing model (*i.e.*, bulk data transfer routines such as `upc_memcpy()`, `upc_memget()`, and `upc_memput()`). Without referring to the performance characteristics of a particular UPC platforms, it is invalid to discuss which design be chosen for a program. Given a particular UPC platform and its performance characteristics, then the performance model can serve as a useful tool in evaluating performance.

One can broadly divide UPC platforms into four types by looking at them along two dimensions: Remote access latency and access pattern awareness. Table 8.3 lists the four types, along with the programming styles suitable to each type and the trade-offs need to be considered.

MuPC with caching disabled is a typical example of the first type platform. As shown in Table 8.3, the programming style that favors performance on this platform stresses scalar access aggregation.

Platform types	*Favorable programming style*	*Trade-offs*
High remote access latency, access pattern unexploited	Minimize the number of remote accesses. Aggregate scalar accesses into bulk data transfers.	A compromise of programmability. Performance on a par with MPI.
High remote access latency, access pattern exploited	Minimize the number of scattered, random accesses. Maximize the number of cache-friendly and vectorizable accesses.	Better programmability. Performance not as good as that of the coarse-grain style.
Low remote access latency, access pattern unexploited	Aggregate scalar accesses into bulk data transfers.	Even better programmability. Performance not portable to high latency systems.
Low remote access latency, access pattern exploited	Maximize the number of cache-friendly and vectorizable accesses.	Best programmability, best performance. Performance not portable to other platforms.

Table 8.3: Recognizing favorable programming styles with respect to a UPC platform

The impact of programing style choice can be illustrated by two matrix multiply implementations: one is the straightforward implementation that uses only fine-grain scalar accesses, the other aggregates remote shared reads using `upc_memget()`. The first implementation takes 179 seconds to multiply two matrices with 256×256 doubles each using 4 UPC threads, while the second implementation needs only 1.7 seconds, a difference of two orders of magnitudes!

8.4 Future work

The theme of this work is performance analysis and UPC system design. It can be easily extended on both sides.

8.4.1 Performance analysis

This research focuses on the performance of fine-grain shared memory accesses, based on the assumption that those are very critical to good performance. The work should be complemented with performance analyses of other operations, notably the collectives, the bulk data transfer operations, and the synchronization operations. These operations are widely used in parallel applications. They are even more important on platforms with low remote access latencies and that are incapable of exploiting access patterns. As has been done in this work, a set of microbenchmarks and models to characterize these operations should be developed.

Most this work relies on microbenchmarks. Microbenchmarks measure discrete fine-grain operations. On the other hand, high performance computing study has been heavily utilizing application benchmarks, for example, the LINPACK benchmark and the NAS Parallel Benchmark suite for distributed memory platforms, and the SPLASH and the SSCA benchmarks for shared memory

platforms, but none of these is suitable or available for UPC. So a major extension of this research should be the development of a UPC-friendly application benchmark suite. A good starting point could be a UPC implementation of the SSCA application kernels [BM05]. The SSCA kernels are a set of graph algorithms for large, directed, weighted multi-graphs. They were originally designed for symmetric multiprocessors, but they are good candidates for exploring UPC's capability for irregular, fine-grain memory accesses.

Other potential extensions to this research include a thorough scalability study. Most results in this work are obtained on architectures with only a few processors. To make these results more valuable in practice, the experiments and analyses should be extended to real-world architectures with hundreds or thousands processors. Shared memory contention will become a non-negligible issue in such environments. Performance models need to account for this additional factor.

8.4.2 Runtime system/compiler design

The work on MuPC can be extended to achieve a more efficient and more sophisticated UPC implementation. First, a compiler middle-end capable of source-level static analysis is very desirable. The current compiler is only a juxtaposition of the EDG front end and an ANSI C compiler. Many optimizations envisioned in this research, such as remote access aggregation and prefetching, however, can only be done through static analysis.

Second, porting MuPC to large-scale parallel systems would be a valuable project. MuPC was mostly developed, tested, and benchmarked on Beowulf-like clusters with a small number of processors. It is unclear how it would behave and whether it scales on larger clusters or on other types of large-scale multiprocessors. It is also worthwhile to re-implement MuPC's communication layer using MPI-2, tapping the benefits of one-sided communication and parallel I/O provided by MPI-2.

Third, the remote reference caching scheme used by MuPC today is overly rigid. For sophisticated applications that have different access patterns in different stages the current caching scheme may improve the performance of some stages but hurt that of other stages. An adaptive caching scheme is envisioned, where enabling and disabling caching is controlled dynamically by execution flow based on remote memory accessing trace collected by runtime during the execution.

Finally, the issue of fault tolerance in UPC is so far largely ignored by researchers. In any of today's implementations, a UPC thread is always scheduled to one and only one processor. Once started, a UPC thread does not migrate and cannot be restarted. This does not guarantee robustness on real-world multiprocessors. Multiprocessor systems built with COTS components are the mainstream of today's high performance parallel systems. These systems usually suffer from a high failure rate. For long-running programs written in UPC, it is critical to protect them with fault tolerance and/or recovery mechanisms.

Chapter 9

Summary

This research accomplishes on two fronts: UPC performance modeling and runtime system implementation.

The major contribution on the first front is performance modeling methodology proposed for PGAS languages and validated with respect to UPC. This is the first effort of understanding the performance phenomena of PGAS languages through performance modeling. This methodology is realized using a set of microbenchmarks and an algorithm for characterizing shared references. In particular, UPC platform benchmarking and the dependence analysis for shared references employed in this work allowed accurate prediction of UPC application performance and reliable characterization of UPC platforms.

On the second front, the major achievement is the MuPC runtime system, which has been used as a tool by this and other researchers to study performance issues and to develop new UPC language features. MuPC is a UPC implementation based on MPI and the POSIX threads. It offers performance comparable with other open source implementations. The implementation work is accompanied with a careful investigation on remote reference caching. The results helped identify loopholes in the current UPC implementations, and provide insights for future performance improvements.

Bibliography

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model: one step closer towards a realistic model for parallel computation. In *SPAA '95: Proc. of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM Press, 1995.

[AK02] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[BB03] C. Bell and D. Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of Workshop on Communication Architecture for Clusters*, 2003.

[BBB94] D. Bailey, E. Barszcz, and J. Barton. The NAS Parallel Benchmark RNR. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BBC+03] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An Evaluation of Current High-Performance Networks. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2003.

[BD03] D. Bonachea and J. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal on High Performance Computing and Networking*, 2003.

[Ber66] A. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, 15:757–763, 1966.

[BGW92] Eugene D. Brooks, III, Brent C. Gorda, and Karen H. Warren. The Parallel C Preprocessor. *Scientific Programming*, 1(1):79–89, 1992.

[BHJ+03] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, W. Pugh, P. Sadayappan, J. Spacco, and C-W. Tseng. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures. In *Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[BM05] D. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings, 12th International Conference on High Performance Computing (HiPC 2005)*, 2005.

[CBD+03] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of 17th Annual International Conference on Supercomputing (ICS)*, 2003.

[CD95a] W. Carlson and J. Draper. Distributed Data Access in AC. In *Proceedings of 5th ACM SIGPLAN Symposium on PPOPP*, July 1995.

[CD95b] William W. Carlson and Jesse M. Draper. AC for the T3D. In *CUG 1995 Spring Proceedings*, pages 291–296, Denver, CO, March 1995. Cray User Group, Inc.

[CDC+99] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, ACM/IEEE Conference on Supercomputing (SC 1993): Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[CEG02] F. Cantonnet and T. El-Ghazawi. UPC Performance and Potential: A NPB Experimental Study. In *Proceedings, ACM/IEEE Conference on Supercomputing (SC 2002): Baltimore, Maryland*, November 2002.

[CG04] K. Cameron and R. Ge. Predicting and Evaluating Distributed Communication Performance. In *Proceedings, ACM/IEEE Conference on Supercomputing (SC 2004): Pittsburgh, Pennsylvania*, 2004.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *PPOPP '93: Proc. of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM Press, 1993.

[CKY03] W. Chen, Arvind Krishnamurthy, and K. Yelick. Polynomial-time Algorithms for Enforcing Sequential Consistency in SPMD Programs with Arrays. In *Proceedings of 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[Cra03] Cray Inc. *Cray X1 System Overview*. Cray Inc., 2003. http://www.cray.com/craydoc/manuals/S-2346-22/html-S-2346-22/z1018480786.html.

[CS03] Kirk W. Cameron and Xian-He Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *IPDPS '03: Proc. of the 17th International Symposium on Parallel and Distributed Processing*, page 48.2. IEEE Computer Society, 2003.

[CYA+04] F. Cantonnet, Y. Yao, S Annareddy, A. Mohamed, and T. El-Ghazawi. Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2004.

[DDH97] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient External Memory Algorithms by Simulating Coarse-Grained Parallel Algorithms. In *Proceedings of ACM Symp. on Parallel Algorithms and Architectures*, pages 106–115, 1997.

[Edi05] Edison Design Group, Inc. Compiler Front Ends for the OEM Market, 2005. http://www.edg.com.

[EGC01] T. El-Ghazawi and S. Chauvin. UPC Benchmarking Issues. In *Proceedings of ICPP (2001)*, 2001.

[EGCD03] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC Language Specifications*, October 2003.
http://www.gwu.edu/∼upc/docs/upc_spec_1.1.1.pdf.

[EGCD05] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC Language Specifications*, May 2005.
http://www.gwu.edu/∼upc/docs/upc_spec_1.2.pdf.

[EGCS+03] T. El-Ghazawi, F. Cantonnet, P. Saha, R. Tharkur, R. Ross, and D. Bonachea. UPC-IO: A parallel I/O API for UPC, 2003.
http://www.gwu.edu/∼upc/docs/UPC-IO_v1pre10.pdf.

[Geo04] George Washington University. Unified Parallel C Home Page, 2004.
http://hpc.gwu.edu/∼upc.

[Geo05] George Washington University. The 5th UPC Developers Workshop, September 2005.
http://www.gwu.edu/ upc/upcworkshop05/agenda.html.

[GV94] A. Gerbessiotis and L. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *J. of Parallel and Distributed Computing*, 22:251–267, 1994.

[H*et al.*01] P. Hilfinger and *et al.* Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, November 2001.

[Hew04] Hewlett-Packard. Compaq UPC for Tru64 UNIX, 2004.
http://www.hp.com/go/upc.

[HHS+95] Chris Holt, Mark Heinrich, Jaswinder P Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical report, 1995.

[HK96] S. Hambrusch and A. Khokhar. C^3, An Architecture-independent Model for Coarse-grained Parallel Machines. *Journal of Parallel and Distributed Computing*, 32, 1996.

[IFH01] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: a parallel computational model for synchronization analysis. In *PPoPP '01: Proc. of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 133–142. ACM Press, 2001.

[IHC04] C. Iancu, P. Husbands, and W. Chen. Message Strip Mining Heuristics for High Speed Networks. In *Proceedings of VECPAR*, 2004.

[Int04] Intrepid Technology. Intrepid UPC Home Page, 2004.
http://www.intrepid.com/upc.

[ISO00] ISO/IEC. *Programming Languages - C, ISO/IEC 9989*, May 2000.

[KW04] W. Kuchera and C. Wallace. The UPC Memory Model: Problems and Prospects. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2004.

[LCW93] James R. Larus, Satish Chandra, and David A. Wood. CICO: A Practical Shared-Memory Programming Performance Model. In Ferrante and Hey, editors, *Workshop on Portability and Performance for Parallel Processing*, Southampton University, England, July 13 – 15, 1993. John Wiley & Sons.

[LDea05] P. Luszczek, J. Dongarra, and et al. Introduction to the HPC Challenge Benchmark Suite. In *Proceedings, ACM/IEEE Conference on Supercomputing (SC 2005): Seattle, Washington*, November 2005.

[McC06] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers, 2006.
http://www.cs.virginia.edu/stream/.

[MF98] Csaba Andras Moritz and Matthew I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. In *SIGMETRICS '98/PERFORMANCE '98: Proc. of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 254–263. ACM Press, 1998.

[Mic05] Michigan Technological University. UPC Projects at MTU, 2005.
http://www.upc.mtu.edu.

[NR98] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN FORTRAN Forum*, 17(2):1–31, August 1998.

[PGA05] The First PGAS Programming Models Conference, September 2005.
http://www.ahpcrc.org/conferences/PGAS/abstracts.html.

[PHP+03] J. Prins, J. Huan, B. Pugh, C-W. Tseng, and P. Sadayappan. UPC Implementation of an Unbalanced Tree Search Benchmark. Technical Report 03-034, Department of Computer Science, University of North Carolina, October 2003.

[Ree03] D. A. Reed, editor. *Workshop on the Roadmap for the Revitalization of High-End Computing*. Computing Research Association, 2003.

[Sav02] J. Savant. MuPC: A Run Time System for Unified Parallel C. Master's thesis, Department of Computer Science, Michigan Technological University, 2002.

[SG98] M. Snir and W. Gropp. *MPI: The Complete Reference*. The MPI Press, 2nd edition, 1998.

[SLB+06] H. Su, A. Leko, D. Bonachea, B. Golden, H. Sherburne, M. Billingsley III, and A. George. Performancerallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming Models. In *Proceedings, ACM/IEEE Conference on Supercomputing (SC 2006) Poster Session: Tampa, Florida*, 2006.

[SS04] E. Strohmaier and H. Shan. Architecture Independent Performance Characterization and Benchmarking for Scientific Applications. In *Proceedings of the 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 467–474, 2004.

[SS05] E. Strohmaier and H. Shan. Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms. In *Proceedings, ACM/IEEE Conference on Supercomputing (SC 2005): Seattle, Washington*, November 2005.

[Tan01] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[UC 04a] UC Berkeley. Berkeley Unified Parallel C Home Page, 2004. http://upc.nersc.gov.

[UC 04b] UC Berkeley. GASNet Home Page, 2004.
http://www.cs.berkeley.edu/∼bonachea/gasnet.

[WGS03] E. Wiebel, D. Greenberg, and S. Seidel. UPC Collectives Specification 1.0, December 2003.
http://www.gwu.edu/∼upc/docs/UPC_Coll_Spec_V1.0.pdf.

[ZS05] Z. Zhang and S. Seidel. Benchmark Measurements for Current UPC Platforms. In *Proceedings of IPDPS'05, 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[ZSS06] Z. Zhang, J. Savant, and S. Seidel. A UPC Runtime System based on MPI and POSIX Threads. In *To appear in the Proceedings of 14th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2006)*, 2006.