A Performance Model for Fine-Grain Accesses in UPC

Zhang Zhang and Steven R. Seidel

Michigan Technological University
Dept. of Computer Science
Houghton, MI 49931-1295 USA
zhazhang, steve@mtu.edu

Abstract

UPC's implicit communication and fine-grain programming style make application performance modeling a challenging task. The correspondence between remote references and communication events depends on the internals of the compiler and runtime system. This correspondence is often hidden from application developers. Aggressive optimizations allowed by the relaxed memory consistency model further blur this correspondence by transforming code structure. A modeling approach based on UPC platform benchmarking and code analysis is proposed. This approach abstracts a UPC platform according to its potential to apply a few common optimizations, then divides remote references in the application code into groups, based on a dependence analysis, that are amenable to each optimization. Each group is associated with a cost, obtained via benchmarking each potential optimization. The aggregated cost of these groups is the predicted cost of the application. Three simple UPC applications modeled using this approach usually yielded performance predictions within 15 percent of actual running times.

1 Introduction

Unified Parallel C (UPC) is an extension of ANSI C for programming multiprocessors [8, 15, 14, 26]. UPC is a member of a family of languages that provide the Parallel Global Address Space (PGAS) programming model [10]. PGAS languages are of interest to the HPC community for their potential of achieving high productivity and high performance, which are essential to the HPCS.

While substantial efforts have gone into PGAS languages design and implementation, little work has been done to model the performance of these languages. At the first PGAS Programming Models Conference [25] and at the 5th UPC Developers Workshop [1], 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.

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 [17] and LogP [11], and many variations [12, 18, 2, 20, 24, 7, 19]. The "check-in, check-out" (CICO) performance model [22] has been proposed for 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.

This paper is the first to address the demand for an application level UPC performance model. This study focuses on the performance of fine-grain shared memory accesses in an execution environment with relaxed memory consistency. We first investigate the UPC PGAS 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.

The remainder of this paper is organized as follows. Section 2 briefly reviews the features of the UPC programming model. Section 3 discusses platform abstraction and application analysis. Microbenchmark design, the principles of

1-4244-0054-6/06/$20.00

relating applications to platform features, and the method of characterizing shared references are also discussed. Section 4 develops equations for run time and speedup prediction. Section 5 reports the results of applying the performance model to three UPC programs run on three different compiler/architecture combinations. Section 7 is a summary of findings.

2 UPC programming model

This section discusses some conspicuous features of the PGAS 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 the 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 distant 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, we take 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.

3 Modeling UPC fine-grain access performance

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, inside knowledge about the particular UPC compiler and runtime system is needed but this is not always available. We 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. We show 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.

3.1 Platform abstraction

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 [5]. 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. We anticipate 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.

```
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] = ...; }
```

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 below.

```
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 sufficient 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.

3.2 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 measures 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 measurement represents 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*. We define $S_{baseline}$, S_{vector}, $S_{coalesce}$ and S_{local} to be the rates of the four patterns.

3.3 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 [6] established the constraints for concurrent scheduling of memory operations. That is, dependence-free accesses can be safely executed in parallel. Following [3], we define dependence 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, we also use the concept of *input dependence*, 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. We define strict operations, including strict memory accesses, barriers, fences, and library function calls such as collectives, to be *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 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 3.2, 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, we 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 times 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.

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.

3.4 Reference partitioning

Reference partitioning can be reduced to a variation of the *typed fusion* problem [3]. We thus define the *reference partitioning* problem 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 such 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 a 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 $e \in B$ 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 are accessing 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:

In the example below 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.

```
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]; }
```

In example below 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.

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

4 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:

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

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. In our experiments the values of $r(N_j, pattern)$ 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 gap between the speed of private memory access and the speed of shared memory access gives an upper bound on how much communication can overlap with computation. Let $S_{private}$ be the private memory access speed, which can be obtained using the existing microbenchmarks such as STREAM [23], and let $S_{baseline}$, S_{vector}, $S_{coalesce}$, and S_{local} be as defined as in section 3.2. Then we define $G_{baseline}$, G_{vector}, $G_{coalesce}$ and G_{local} to be the *gaps* between the shared memory access speeds and the private memory access speed:

$$\begin{aligned} G_{baseline} &= \tfrac{S_{baseline}}{S_{private}}, \quad & G_{vector} &= \tfrac{S_{vector}}{S_{private}}, \\ G_{coalesce} &= \tfrac{S_{coalesce}}{S_{private}}, \quad & G_{local} &= \tfrac{S_{local}}{S_{private}}. \end{aligned} \tag{2}$$

Smaller gaps mean that the shared memory access speed is relatively faster and a shared memory access can overlap with less local computation. For example, if the gap 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 a 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 we can model the speedup achievable by the parallelized code using the ratio N_s/N_p:

$$S = \frac{T_s}{T_p} \approx \frac{N_s}{N_p} = \frac{N_s}{N_c + (N_l \times G_{local} + N_r \times G_{remote})} \quad (3)$$

where T_s is the sequential run time and T_p is the parallel run time prediction based on Equation (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 if it is not orders of magnitude larger 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.

5 Applications of the performance modeling method

In this section we provide some validation of the performance modeling method using three simple UPC applications: 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.

We use the performance model 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:

$$\delta = \frac{predicted\ cost - actual\ cost}{actual\ cost} \times 100\% \quad (4)$$

The experiments were done using MuPC V1.1.2 beta [29, 30, 27], Berkeley UPC V2.2 [4, 9], and GCC UPC V3.4.4 compilers [21]. 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.

Table 1 contains measurements of the four microbenchmarks for the platforms above. Measurements are reported for 2 and 12 threads in units of 10^3 (K) or 1^6 (M) double word accesses per microsecond, so larger is better. The access rates decrease as the number of threads increase. Caching improves MuPC performance for the vector and coalesce benchmarks and it reduces performance for the baseline write benchmark. Berkeley UPC successfully coalesces reads. GCC UPC performs local writes surprisingly slowly. This was also noted in [30].

5.1 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++) {
  // loc is a pre-computed random number
  array[loc]++; }
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. We model 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

Microbenchmark	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC UPC	
(threads)	read	write	read	write	read	write	read	write
baseline (2)	14.0K	35.6K	23.3K	11.4K	21.0K	46.5K	0.45M	1.3M
(12)	12.0K	30.8K	10.8K	7.3K	15.1K	43.3K	0.4M	1.1M
vector (2)	16.4K	45.4K	1.0M	1.0M	21.1K	47.2K	0.5M	1.7M
(12)	14.0K	34.6K	0.77M	0.71M	14.6K	44.8K	0.45M	1.7M
coalesce (2)	16.7K	43.9K	82.0K	69.9K	172K	46.9K	0.5M	1.6M
(12)	12.9K	39.5K	64.1K	46.3K	122K	44.4K	0.4M	1.5M
local (2)	8.3M	8.3M	8.3M	8.3M	8.3M	6.7M	1.2M	0.7M
(12)	8.3M	8.3M	8.3M	8.3M	6.7M	5.0M	1.0M	0.62M

Table 1. Microbenchmark measurements for 2 and 12 threads (double words/microsecond).

accesses, and the effective data transfer rates obtained using the *baseline* and the *local* microbenchmarks. The results in Table 2 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%.

	δ (%)		
Percentage	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 2. Prediction precision for histogramming. The size of the histogram table is $1M$, THREADS $= 12$. The results are averages of at least 10 test runs.

5.2 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 [16]:

```
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;
```

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 these experiments N is always divisible by THREADS):

```
// cyclic striped
shared [N] double A[N][N];
shared     double B[N][N];
shared [N] double C[N][N];

// block 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];
```

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 implementa-

tions 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 3. The negative errors in the cases of Berkeley UPC and GCC UPC show an underestimation of run times for these two platforms. We suspect that there 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. It was also noted that GCC UPC array references with two subscripts are $20-60\%$ more expensive than array references with one subscript. We believe this is a performance bug. The corresponding measurements shown in Tables 3 and 4 take this into consideration.

5.3 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 [16] is shown in the next page.

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 containing references to the `i`-th row of `O`. 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 `O`. (4) A partition containing references to the `(i+1)`-th row of `O`. 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 4. 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.

```
#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; }
}
```

6 Discussion and future work

This model can be improved in several ways. First, it should be expanded to model the performance of coarse-grain shared memory operations such as `upc_memcpy()`, `upc_memput()`, and `upc_memget()`. Second, interactions between memory operations and computation should be more thoroughly studied. Currently, the model predicts run time based on information about memory operations only. Advances in UPC systems will increase overlap between memory accesses and computation. The model should include elements to account for this overlap. Third, memory bandwidth contention, especially for remote accesses, needs to be considered. This is important in analyzing applications with unbalanced communication patterns. Finally, the model should be applied to more complex applications and to systems that support larger numbers of threads. This will give greater weight to the validity and utility of the model for real-world applications.

At some point in the future compilers are expected to identify patterns that are not currently in this model. However, even the patterns discussed here are not all considered by current compilers. Table 1 shows that Berkeley's UPC successfully takes advantage of coalesced reads but no other patterns are exploited, MuPC's runtime cache exploits coalesced and vector accesses, and GCC UPC exploits none of the patterns. Future work will include consideration of the Cray X1 compiler [13] and the HP UPC compiler and runtime system [28] which has a trainable prefetcher and runtime cache.

There are several other metrics that can be considered for inclusion in the model. For example, in systems that support remote access caching, a miss penalty (a platform metric) and a miss rate (an application metric) could be used to investigate cache effects. For compilers that perform prefetching, the prefetch depth can be a useful plat-

THREADS	δ (%)							
	MuPC w/o cache		MuPC w/ cache		Berkeley UPC		GCC UPC	
	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 3. Prediction precision for matrix multiply. The size of matrices are 240×240 (doubles). The results are averages of at least 10 test runs.

THREADS	δ (%)			
	MuPC w/o cache	MuPC w/ cache	Berkeley UPC	GCC UPC
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 4. Prediction precision for Sobel edge detection. The image size is 2000×2000 (integers). The results are averages of at least 10 test runs.

form metric. The number of outstanding shared memory operations supported by a platform can help to evaluate the platform's scalability. These metrics are likely to improve the accuracy of the model but they are not easy to measure and require a more thorough understanding of a platform's internals.

Other issues include the effects of load imbalance and hot spots. Load imbalance may lead to two types of effects in a UPC computation. A computational load imbalance need not necessarily cause hot spots in shared memory. Some threads may have larger work loads while the access pattern across the whole address space remains uniform. This type of load imbalance can be modeled by assigning a weighted computational cost to each thread proportional to its work load if that load is predictable. The other type of load imbalance causes hot spots in memory. In this case, the hot spot can be modeled with a queue where contending threads wait for their memory requests to be served. Adding a queuing model to the performance model may be what is needed to predict the cost in this case. For both types, the difficult task is to identify where and when the imbalance will occur.

7 Summary

This paper proposed an approach to model the performance of UPC programs that have implicit, fine-grain shared memory accesses. This is the first performance modeling methodology proposed for UPC. This approach recognizes four basic reference patterns and accordingly uses four simple microbenchmarks to measure a UPC platform's ability in optimizing fine-grain shared memory accesses. Next, a dependence-based analysis is used to partition references in an application into groups and 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.

The model predicted the run times of three applications running on three different UPC platforms. The prediction has a maximum error of ±15% in most cases. This is a good accuracy for an analytical performance model. Factors that led to inaccuracy were also discussed.

References

[1] 5th UPC Developers Workshop, George Washington University. Sept. 2005. http://www.gwu.edu/~upc/upcworkshop05/agenda.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. 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.

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

[4] Berkeley Unified Parallel C Home Page. UC Berkeley. http://upc.nersc.gov.

[5] 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.

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

[7] K. W. Cameron and X.-H. 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.

[8] 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.

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

[10] W. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-grained UPC Applications. In *Proc. of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. 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.

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

[13] T. El-Ghazawi, F. Cantonnet, Y. Yao, and J. Vetter. Evaluation of UPC on the Cray X1. In *Cray User Group Proceedings*, 2005.

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

[15] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[16] T. El-Ghazawi and S. Chauvin. UPC Benchmarking Issues. In *Proc. of ICPP*, 2001.

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

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

[19] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical report, 1995.

[20] F. Ino, N. Fujimoto, and K. 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.

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

[22] J. R. Larus, S. Chandra, and D. 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.

[23] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[24] C. A. Moritz and M. 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.

[25] PGAS Programming Models Conference, Army High Performance Computing Research Center. Sept. 2005. http://www.ahpcrc.org/conferences/PGAS/.

[26] Unified Parallel C Home Page. George Washington University. http://hpc.gwu.edu/~upc.

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

[28] UPC Version 2.4, Hewlett-Packard Company. 2006. http://www.hp.com/go.upc.

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

[30] Z. Zhang and S. Seidel. Benchmark Measurements for Current UPC Platforms. In *Proc. of IPDPS'05 (PMEO-PDS Workshop), 19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005.