Predicting Remote Reuse Distance Patterns in UPC Applications [*]

Steven Vormwald
Cray, Inc.
380 Jackson St.
St. Paul, Minnesota
sdvormwa@cray.com

W. Wang S. Carr S. Seidel Z. Wang
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931
weiwang,carr,steve,zlwang@mtu.edu

Abstract

Current work in high productivity parallel computing has focused attention on the class of partitioned global address space (PGAS) parallel programming languages because they promise to reduce the effort required to develop parallel application codes. An important aspect in achieving good performance in PGAS languages is effective handling of remote memory references. We extend a single-threaded reuse distance model to predict memory behavior for multi-threaded UPC applications. Our model handles changes in per-thread data size as well as changes in thread mapping due to problem size increases. Our results indicate the model provides good predictions of remote memory behavior by accurately predicting changes in remote memory reuse distance as a function of the problem size and the number of threads.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: parallel programming; D.3.2 [*Programming Languages*]: concurrent, distributed, and parallel languages; D.3.4 [*compilers; optimization*]

General Terms languages, performance, measurement

Keywords UPC, reuse distance, cache

1. Introduction

Reuse distance analysis [4–7, 10] has proved to be a good mechanism to predict the memory behavior of single-threaded programs over varied input sets. The *reuse distance* of a memory reference is the number of distinct memory locations accessed between two references to the same memory location. In essence, data with a reuse distance smaller than the cache size will remain in the cache long enough for the second access to yield a cache hit. Both whole-program [4, 10] and instruction-based [5–7] reuse distance have been predicted accurately across all program inputs based on a few profiling runs.

We define the *remote reuse distance* of a remote memory reference in a PGAS program as the number of distinct remote memory locations accessed between this reference and the next access to the same remote location. Current reuse distance models are only designed for single-threaded programs. If one wishes to consider PGAS applications, these models cannot adequately predict the locality of the threads since the models do not consider the effect of the distribution of threads and data on the communication patterns for any given thread. In order to analyze PGAS applications, we must extend the model to include these phenomena.

To address this issue, we present a heuristic that accurately predicts thread behavior when changes to the problem size cause the distribution of threads in the problem domain to change. For a set of representative UPC kernels, our experiments show that we can often achieve greater than 90% reuse distance prediction accuracy using our heuristics to map threads from small problems sizes to large problem sizes. Using our prediction scheme allows us to predict the remote memory behavior of many UPC application domains and predict the remote memory performance of those applications.

2. Remote Reuse Prediction Through Thread Matching

We predict remote reuse distances for each instruction in each thread. Borrowing the notations used in the SPEC CPU benchmarks and related literature for easy discussion, we use the `test` and `train` runs to denote the two training runs used for profiling. Based on the training runs, we predict remote reuses in the `reference` run. The `reference` run typically has a larger data size. We consider two possibilities for data size increases. First, the number of threads does not change in the `test`, `train` and `reference` runs while the data set size in each thread increases, which results in a problem size increase. Second, both the local data set size and the number of threads increase.

Since a prediction is being performed for each UPC thread, the choice of which threads in the `test` and `train` are used in predicting a particular thread in the `reference` run is critical. If the number of threads is kept constant, the obvious choice is to use the thread from each training run with the same thread id to predict the corresponding `reference` thread. This prediction is identical to the single-thread prediction case.

The challenge is when the problem size scales along with the number of threads. No matter whether the data size for each thread remains the same or not, there is no longer a one-to-one correspondence among the thread ids in the three runs (*i.e.*, the `test`, `train` and `reference` runs). Note that the communication pattern is typically based on the data distribution across the threads. The communication pattern therefore is connected with the geometric layout of the data.

[*] This research has been partially supported by NSF Grants CCF-0833082, CCF-0811427, and CCF-063664.

We designed an algorithm that identifies this geometric layout, and then chooses threads for prediction based on it. As an exception, thread 0 is assumed to be used for various unique tasks, such as initialization, and is therefore always predicted with thread 0 from each of the training runs. Our approach is divided into three steps. First, independently, in the `test` and `train` runs, we group the threads based on the similarity of remote reuse patterns. The second step compares the grouping results and chooses a *geometric pattern* function that best fits the grouping pattern from step 1. Lastly, in the third step, we apply the geometric pattern function to thread groups in the `reference` runs for prediction.

After each training run, we collect remote reuse distance data for each instruction in each thread and generate reuse patterns following the algorithm proposed by Fang *et al.* [6]. Our prediction is instruction based. For each memory access instruction, we group the threads based on the reuse patterns. Intuitively, two threads are put into the same group if they show the same communication pattern. The partitioning algorithm admits a thread into a group if the thread's pattern matches the group pattern. The group pattern summarizes reuse for all threads in this group where each group pattern takes the arithmetic average of the corresponding reuse patterns in the thread group. We say the reuse patterns of an instruction in a thread *matches* the group pattern if all reuse locality patterns match by at least 95%. After this step, in each run, all threads that show similar remote access pattern are grouped together. We observe that this partitioning step automatically identifies the program behavior in terms of its communications patterns without input from the programmer or compiler analysis.

Step 1 partitions the threads into groups, Step 2 then matches these groups against a geometric description function that describes the expected partitioning of threads. The geometric pattern function is used to generate the set of values associated with the geometric location of the threads in each group. Shared data are distributed across the threads following certain geometric patterns depending on the problem itself and the algorithm the programmer chooses to solve the problem. Our current approach can handle five out of seven types of parallel applications as categorized by Berkeley [2]. In these applications, the programmer views the data layout across the threads in a two-dimensional, three-dimensional, or hyper-cube manner which dominates typical data layouts in parallel applications. We have thus designed a set of functions that summarize geometric partitions in these layouts.

As an example, the pattern function for the two-dimensional view is shown in Figure 1. We envision the threads along the edges, on the diagonal line and in the corners might show distinct communication patterns. Note that the pattern function partitions a square into 11 distinct regions based on geometric properties. A simple algorithm maps the partitioned thread groups in Step 1 into these regions. A group could cover multiple regions and thus the regions of a group are a subset of $\{0,1,2,\ldots,10\}$.

Given an instruction, our algorithm does not need to know the data distribution it accesses to find the pattern function. We instead test it against a library of functions to find a fit. If none of the functions fit the instruction we cannot predict reuse for the instruction. Finally, if both training runs match the pattern, threads are predicted with threads from the training runs that share the same group as determined by the matched function.

3. Experimental Evaluation

All the experiments are performed on a 24-node dual-core Intel Xeon cluster. The instrumentation is done with a modified version of the MuPC compiler [8]. We insert callback functions into UPC runtime functions, and retrieve the program's remote access information, such as the return address of remote access operation, destination thread ID, remote address, and calling context. The information is then fed into a reuse distance collection tool, which is adapted from the work of Fang et al. [6]. After the instrumentation, the remote reuse distance data is organized as histograms on a per PC/per thread basis. In our experiment, we use a 64-byte data block as the smallest data unit.

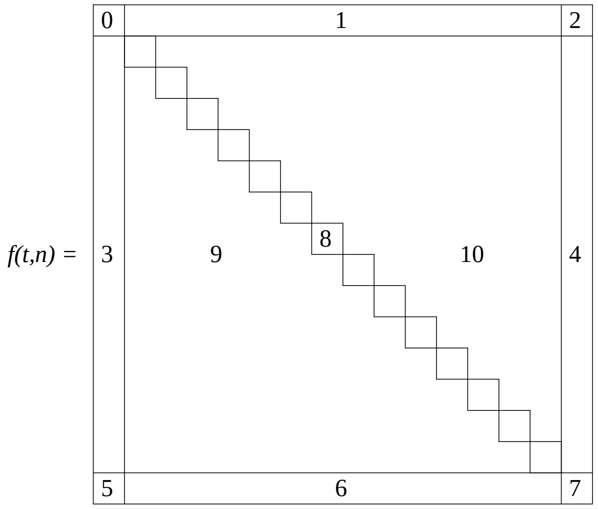

Figure 1. Geometric Pattern Function

We evaluate six kernels with a number of different data sizes and number of threads. Since there are no standard benchmark applications for UPC, the six kernels were written or modified from existing applications. Table 1 gives a brief description of each kernel, as well as the local problem sizes and number of threads we choose.

Due to hardware and software limitations, testing is restricted to a maximum of 125 UPC threads. Most results are for fewer than 24 threads. While we recognize that these are relatively small systems in the world of high performance computing, we believe that the results will hold as the problem size and number of threads increase.

There are two important measures of reuse distance prediction. First is the *coverage*, which is defined as the percentage of operations in the reference run that can be predicted. An operation can be predicted if it occurs in both training runs, and all of its patterns are *regular*. A pattern is *regular* if it occurs in both training runs and the reuse distance does not decrease as the problem size grows.

The *accuracy* then is the percentage of *covered* operations that are predicted *correctly*. An operation is predicted correctly if the predicted patterns exactly match the observed patterns, or they overlap by at least 90%. The overlap for two patterns A and B, where $B.min < A.max \leq B.max$, is defined as

$$\frac{A.max - \max(A.min, B.min)}{\max(B.max - B.min, A.max - A.min)}$$

The overlap factor of 90% was used in this work because this factor worked well in prior work with sequential applications [6].

Given the set of runs for a kernel with different problem sizes and number of threads, we pick any three runs that meet the criteria of either problem size scaling or thread scaling. The two training runs are used to predict the remote reuse patterns in the `reference` run. The minimum, maximum, and average prediction accuracy and coverage of all combinations are measured for each kernel. (Since the measured maximum accuracy and covereage was always at least 99%, these data do not appear in the tables.)

Name	Description	Problem Size		Thread Number	
		Min	*Max*	*Min*	*Max*
Matrix Multiply	Multiplication of two dense square arrays. each thread has an NxN block of the array	4	256K	4	16
Jacobi	A Jacobi iterative solver for a large array distributed in a block cyclic fashion	2	8K	4	24
LU 8X8/16X16/32X32	LU Decomposition on a large array, from test suite from Berkeley's UPC compiler [1]. Square blocks of 8X8,16X16 or 32X32 are distributed to each thread in round-robin fashion	1024	16M	2	36
2D Stencil-4p/8p	Apply a 4-point or 8-point 2D Stencil to a 2D array. Each thread is assigned a sub-array of size NxN.	64	16M	9	49
3D Stencil-6p/26p	Apply a 6-point or 26-point 3D Stencil to a 3D array. Each thread is assigned a sub-array of size NxNxN.	64	16M	27	125
3D FFT	Apply Fast Fourier transform using a square 3D array Cooley-Tukey algorithm. Each thread is assigned a sub-array of size NxNxN.	512	256K	4	64

Table 1. Kernel Description

3.1 Problem Size Scaling

We measure our model under problem size scaling with a fixed number of threads while increasing problem size. The remote reuse patterns of thread *i* in the `reference` run are predicted using the patterns from thread *i* in the two training runs.

Table 2 shows that both coverage and accuracy are quite high in most cases. There are two main factors that lead to low performance in some predictions. The first factor is using training runs that are too small to predict the larger `reference` run. For example, of the 1612 predictions on the matrix multiplication kernel where the accuracy is less than 60%, 1610 of them occur when the smallest training size is 256 or less and 1332 occur when the smallest training size is 16 or less. For the stencil kernels, while the 2D 4-point stencil and the 3D 6-point stencil do not change much when the problem size changes, the 2D 8-point stencil and the 3D 26-point stencil do suffer from low accuracy and coverage when the smallest problem size is chosen to be one of the training runs. The reason is that patterns in the 4-point and 6-point stencils are much simpler than those in the 8-point and 26-point stencils. For example, there are up to 4 additional remote reference operations in the 8-point stencil than the 4-point stencil. Each thread requests remote data from 3 neighboring threads, thus having more complicated patterns. When the problem size is too small, some details of the patterns are missing in those operations, resulting in low accuracy and coverage. In the FFT kernel, all accuracies below 70% occur when selecting the two smallest training runs. Thus, in our analysis, training data sets must be chosen to be large enough to exhibit the characteristics of the larger runs.

The second factor shows its impact when the growth of the problem size causes the distribution of shared data to change, which in turn causes the communication pattern between threads to change. This behavior is seen in the LU kernel, where prediction accuracy and coverage drop steeply in a few cases because the distribution of the array changes due to changes in the problem size. The data distribution in turn determines which remote memory operations a thread encounters. When a large number of remote memory operations in the training runs does not match the reference run, prediction suffers low coverage. In fact, when the number of remote memory operations in the three runs can be matched, the average coverage for LU 8X8, 16X16 and 32X32 jumps to 98%, 96%, and 99% respectively.

3.2 Thread Scaling

We measure remote reuse distance prediction under thread scaling by varying both the number of threads and the per-thread data size. Table 3 shows the prediction results using all possible pairs of training threads from the training data available. Due to random matching, both coverage and accuracy vary significantly case by case. In the matrix multiplication and Jacobi kernels, the low minimum accuracies are seen when training with very small problem sizes, the same phenomenon that occurs when scaling the problem size. These results are for exhaustively predicting every thread in the reference set with every possible combination of threads in the two training sets. The high average accuracy and coverage in the matrix multiplication and Jacobi kernels in the table indicate that the prediction works well regardless of the threads chosen for training for the two kernels.

Kernel	Accuracy		Coverage	
	Min	*Avg*	*Min*	*Avg*
Matrix Multiply	3%	97%	53%	99%
Jacobi	100%	100%	39%	100%
LU 8x8	6%	94%	0%	77%
LU 16x16	9%	96%	0%	63%
LU 32x32	0%	96%	0%	53%
2D Stencil-4p	99%	100%	100%	100%
2D Stencil-8p	89%	100%	71%	99%
3D Stencil-6p	91%	99%	100%	100%
3D Stencil-26p	9%	83%	58%	90%
3D FFT	60%	88%	100%	100%

Table 2. Problem Size Scaling Accuracy and Coverage

Kernel	Accuracy		Coverage	
	Min	*Avg*	*Min*	*Avg*
Matrix Multiply	3%	98%	28%	99%
Jacobi	94%	100%	56%	91%
LU 8x8	13%	95%	0%	42%
LU 16x16	20%	92%	0%	37%
LU 32x32	13%	82%	0%	21%
2D Stencil-4p	100%	100%	0%	50%
2D Stencil-8p	0%	69%	0%	61%
3D Stencil-6p	95%	100%	0%	50%
3D Stencil-26p	7%	69%	8%	66%
3D FFT	61%	92%	24%	89%

Table 3. Thread Scaling Accuracy and Coverage without Partitioning

The prediction coverage and accuracy on the other kernels are more dependent on the choice of threads used for the training, however. Low accuracy and coverage usually occur when blindly picking two threads that exhibit different communication behaviors in the training runs. Table 4 shows the results of using the thread partitioning and selection operations to select training threads for all kernels. As expected, the coverage and accuracy both show marked improvement for all kernels, especially for average coverages, 6 out of 10 kernels are able to achieve more than 100% improvement. In the LU kernel, the threads that contain blocks on the diagonal have to do additional communication, their remote access patterns differ from other threads. In the stencil kernels, threads working at corners, edges, and middle issue different numbers of remote access operations. Some of the operations may not appear in some threads due to no communication on one or more sides of the stencil. By grouping threads with similar behavior together, and performing prediction based on the grouping, we are able to achieve significant improvement in minimum accuracy, as well as prediction coverage. For the FFT kernel, although every thread communicates with all other threads, thread partitioning suggests grouping threads into pairs. With such partitioning, all the threads can be fully covered.

Kernel	**Accuracy**		**Coverage**	
	Min	*Avg*	*Min*	*Avg*
Matrix Multiply	92%	99%	100%	100%
Jacobi	95%	100%	100%	100%
LU 8x8	51%	91%	100%	100%
LU 16x16	56%	80%	97%	99%
LU 32x32	49%	92%	98%	99%
2D Stencil-4p	100%	100%	100%	100%
2D Stencil-8p	100%	100%	100%	100%
3D Stencil-6p	100%	100%	100%	100%
3D Stencil-26p	100%	100%	100%	100%
3D FFT	69%	93%	100%	100%

Table 4. Thread Scaling Accuracy and Coverage with Partitioning

Table 4 indicates that for the LU kernel the accuracy decreased when thread matching is used. While this may seem counter-intuitive, we can explain it by examining the coverage statistics. When no thread matching is used, our prediction yields high accuracy on only a small percentage of the communication operations. Random thread matching yields low coverage. When we use thread matching, we predict for a much higher percentage of remote references. Thus, even though our accuracy is lower, it is actually much superior to the random matching case since we predict for significantly more remote references.

4. Previous Work

Zhang, *et al.* have developed a performance model for UPC [9]. The performance of a UPC program is determined by platform properties and application characteristics. While this model predicts remote-memory performance, it does not consider the effects of changes in data size and thread layout. Another performance model that specifically targets PGAS languages was introduced by Bakhouya, *et al.* [3]. This model counts the number of computation and communication operations in each phase and distinguishes remote shared accesses from other memory accesses. This model does not consider machine-dependent factors or compiler and runtime system optimizations. In its current form this model is only useful for comparing algorithm designs and data distributions.

5. Conclusion

This paper presents a model for predicting the remote memory reuse patterns in PGAS application programs written in UPC. The model extends current approaches to predicting single-threaded reuse patterns by providing a heuristic to map threads from the profile data to larger problem sizes using additional threads. The model often achieves an accuracy of greater than 90% for several different classes of parallel problem domains.

In the future, we will apply our prediction scheme to software-cache optimization and performance modeling. This scheme will allow us to tailor software cache designs to specific problems and to optimize cache performance based upon reuse patterns. We plan to extend this model to predict reuse distance for more parallel problem domains and to predict fine-grained spatial reuse distance.

References

[1] The Berkely UPC Compiler, 2009.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] M. Bakhouya, J. Gaber, and T. El-Ghazawi. Towards a Complexity Model for Design and Analysis of PGAS-Based Algorithms. In *Lecture Notes in Computer Science (HPCC 2007)*, pages 672–682. Springer-Verlag, September 2007.

[4] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, California, June 2003.

[5] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2nd ACM Workshop on Memory System Performance*, pages 60–68, Washington, D.C., June 2004.

[6] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *Proceedings of the Fourteenth International Conference on Parallel Architecures and Compilation Techniques*, St. Louis, MO, September 2005.

[7] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, June 2004.

[8] 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 (PDP 2006)*, February 2006.

[9] Z. Zhang and S. Seidel. A performance model for fine-grain accesses in UPC. In *Proc. of 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.

[10] Y. Zhong, S. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–101, New Orleans, LA, September 2003.