Tool-assisted performance measurement and tuning of UPC applications

Guojing Cong   Hui-fang Wen
IBM T.J. Watson Research Center
1101 Kitchawan road, Yorktown Heights, NY, 10598
{gcong,hfwen}@us.ibm.com

Yasushi Negishi, Hiroki Murata
IBM Research - Tokyo
1623-14 Shimotsuruma, Yamato-shi,Japan
{negishi,mrthrk}@jp.ibm.com

Abstract

The PGAS paradigm provides a shared-memory abstraction for programming distributed-memory machines. UPC, one of the more popular PGAS languages, improves ease of programming for the user, yet it also makes it difficult for performance analysis to correlate runtime behavior to program constructs. As efficient remote memory access is critical to performance, understanding the communication pattern can bring insight for performance diagnosis and tuning.

In our study we develop a light-weight tracing mechanism to track remote accesses in UPC programs and correlate them to the program structures. Based on the profiling results, we also propose caching and access coalescing through automated code refactoring to improve communication efficiency for irregular codes. We analyze the communication performance of UPC applications on a cluster of SMPs, and show our tool-assisted optimizations achieve significant speedups over the original programs.

1. Introduction

The partitioned global address space (PGAS) paradigm was proposed as a productivity feature for high performance computing systems. PGAS languages such as UPC [13] and X10 [1] present a shared-memory abstraction for programming distributed-memory machines. They give the programmer the control of data layout and work assignment. PGAS languages improve ease of programming and also provide leverage to programmer to tune for high performance.

Cong *et al.* [3] showed that with PGAS mapping shared-memory algorithms onto distributed-memory machines is straightforward. It is also shown in [3], however, that such implementation can be very inefficient for large-scale irregular problems due to the communication cost of numerous remote accesses. The straightforward PGAS performance per processor is much lower than that of the SMP implementation for the connected components algorithm [10]. The performance gap suggests that analyzing remote memory accesses is critical to tuning PGAS programs.

Analyzing remote memory accesses requires correlating runtime communication with high-level program structures to formulate bottleneck hypotheses. Attributing communication cost incurred by implicit, remote accesses to source structures can be convoluted for PGAS programs. The difficulty results from both the runtime layer that is transparent to the user and the compilation scheme. The runtime system typically simulates a virtual shared-memory machine with one-sided communications. Without a standard interface, it is cumbersome or even impossible to capture the runtime communication behavior. In addition, most current compilers invoke two phases of compilation for a PGAS program. First the PGAS program (e.g., UPC program) is compiled into a message-passing paradigm, that is, a native program (e.g., C program) plus runtime communication support (e.g., MPI). Note that the native program may not be actually produced; instead, it may exist only as a compiler intermediate representation. Next, the native program is compiled with a native compiler (e.g., XLC or gcc) into a binary executable. Thus with no special bookkeeping during compilation and runtime, most observed behavior can only be traced back to the native program. Currently several PGAS languages have been proposed. As UPC is arguably the most widely used with the most mature compilers and runtimes, we analyze and tune UPC programs in our study.

In our study we develop a mechanism to trace remote memory accesses in UPC applications. Different from other interfaces, for example, GASP [12], our design closely couples with the compiler and runtime design. As a result, the tracing is light-weight and powerful, and our tool is able to map communication to source code constructs such as functions, lines, and shared data structures. We show that performance analysis assisted with our tool can help the user better understand various aspects of the remote memory accesses in UPC programs.

In addition to assisting performance analysis, our tool also provides automated refactoring support for optimizing remote accesses in *upc_forall* loops through caching and coalescing. We show that our refactorings combined with the insights gained from profiling can help the user detect and alleviate the communication inefficiency resulted from a large number of remote memory accesses. Our experiments show that optimizations assisted by our tool can achieve between 5 to 60 times speedups for the programs that we study.

The rest of the paper is organized as follows. Section 2 discusses prior related work in profiling and optimizing PGAS applications. Section 3 presents our tracing methodology, and analyzes the remote access behavior of a UPC application. We present our automated refactoring support to improve communication efficiency in section 4 and experimental results in section 5. In section 6 we give our conclusion and future work.

2. Related work

GASP [12] is a framework for the implementation of the PGAS programming model to interact with the performance analysis tool. It specifies an event-based interface so that a performance analysis tool can analyze PGAS program performance by defining callback functions for corresponding events. GASP relies exclusively on the compiler for instrumentation and tracing. Although GASP tools are portable (they in theory work with any compiler that implements GASP), the disadvantage of GASP is the increased complexity of compiler design and possibly runtime overhead, especially for shared data structure analysis with compilers that adopt the multi-phase compilation scheme.

GASP-based tools in theory may be able to correlate runtime remote accesses to source *data structures*, yet we are not aware of any study that present such results. For example, a recent performance study of PGAS applications can be found in [11] where a GASP tool is used. The communication is not mapped to the shared data structures.

GASP is not supported by the IBM compiler for any of its high-performance platforms. Another interface similar to GASP is proposed in [5].

Mohr *et al.* [8] proposed a performance measurement infrastructure for Co-array Fortran on Cray X1. The infrastructure uses high-overhead source-to-source instrumentation to intercept remote memory accesses. Although the communication patterns among processors are captured, mapping them to the source is not discussed in [8].

Communication efficiency is a key focus for the compiler and runtime optimization. In [2] Chen *et al.* presented the optimizations that the Berkeley UPC compiler implemented and their performance impact on various kernels. For highly irregular codes as the ones we will study in this paper, the proposed optimizations do not appear to occur.

3. Analyzing remote memory accesses

Statistics about remote accesses, such as the percentage of time spent on communication, the number of remote accesses between two threads, and the amount of data transferred, are helpful for the user to interpret observed performance and postulate bottleneck hypotheses. Correlating the runtime statistics to the source code and the data structures can further help the user focus her optimization efforts. In this section we present the design of our tool to capture remote access statistics, and we use our tool to analyze an example UPC program.

3.1 Profiling methodology

Our tracing tool closely couples with our UPC compiler/runtime [14], yet it does not rely on the compiler for code instrumentation. We base the design of our tool on the internals of the compiler, and our approach is capable of establishing the link between runtime behavior and static structures. The disadvantage is that our tool assumes a certain UPC compilation scheme (e.g., remote accesses are translated into runtime function calls) that may not be adopted by other UPC compilers. To interact with other performance tools, a translation layer may be added to produce the GASP event traces.

With the XL UPC compiler, a runtime *transport* function is invoked for each remote memory access. Tracing remote memory accesses is done by intercepting the transport functions through the *weak symbol* mechanism. Suppose the *remote_get* function fetches data from a remote node. In the runtime *remote_get* is defined as a weak symbol as follows so the linker may use *_remote_get* if *remote_get* is not defined.

```
#pragma weak remote_get=_remote_get
```

The runtime library defines *_remote_get*, and our profiling library defines *remote_get*. The *remote_get* function captures runtime statistics including time spent, amount of data transferred, source and target of the transfer for *_remote_get* in addition to calling *_remote_get* for the actual data transfer. This mechanism is the same as the standard MPI tracing interface [7]. Hence if the UPC runtime implements such an interface, our tool will be able to work with the corresponding compiler.

Such profiling does not require recompiling the code, and incurs relatively low overhead as no instrumentation is done to local memory accesses. As each transport function specifies a remote node and a shared variable described by a *handle* argument, we can attribute the communication statistics to handles (but not yet to the source).

To identify the source lines associated with the communication, there is the option with our tool to walk the stack frames of a profiled transport function. Stack walk stops at the first ancestor whose call site in the source can be determined. Debugging information from the compiler relative to the UPC source is needed for such mapping.

Extra book-keeping is necessary to determine the shared variables involved in the communication. Recall that *weak symbol* profiling captures communication statistics for each handle. Depending on how the handles are created during runtime, we map them to the shared UPC data structures in two ways.

For shared variables allocated at compile time (e.g., global shared variables), the compiler creates temporary variables during the intermediate translation to store the handles. The naming of the variables contains information that can be used to recover the original data structures in the UPC program. We instrument the binary with *psigma* [9] and intercept the handle allocation routine. The handle value for the remote access and the address of the corresponding temporary variable are captured and inserted into a hash table during execution. At exit, the mapping between the communication to the source variable is established. The conceptual process is as follows: *remote access* $\rightarrow$ *handle* $\rightarrow$ *temporary variable* $\rightarrow$ *UPC variable*.

For variables dynamically allocated with *upc_alloc* or *upc_all_alloc*, their handles do not have associated intermediate variables. We associate remote accesses to the source lines where the variables are allocated. To do so, we first analyze the binary and capture the call sites of each shared memory allocation function. We then assign a unique ID, for example, the corresponding binary address, for each call site. The call sites can be mapped using debugging information to the UPC lines. The binary is then instrumented, and we establish during runtime the link between the handle value and the call site ID. Thus we can map each handle value to the binary address of the allocation call site and then to the source line. Simple parsing of the source line suffices to recover the UPC variable associated with the allocation. Conceptually, the process is as follows: *remote access* $\rightarrow$ *handle* $\rightarrow$ *binary address* $\rightarrow$ *source line* $\rightarrow$ *UPC variable*. Note that the mapping of statically allocated variables can not be handled in this fashion as the allocation code is produced by the compiler and does not map to any meaningful UPC sources.

3.2 Measurement

We present as an example our measuring of the remote accesses in a UPC program using the tool we developed. The program implements the connected components (CC) algorithm as described in [3, 10]. CC takes a sparse graph as the input, and computes the maximal connected subgraphs. We use a random graph of 10000 vertices and 40000 edges as the input. We test on a cluster of SMPs with 2 IBM Power5 nodes. We use 4 processors per node. As we do not measure the absolute performance, we defer the introduction of the system to section 5.

In the first run, we turn on only remote access tracing without mapping to the source code. The tracing overhead is low (within 5% of total execution time). The experiment shows that CC spends a significant portion (between 35% and 55%) of running time on remote accesses.

Fig. 1 shows the number of remote accesses on each thread, and Fig. 2 shows the corresponding remote access time as a percentage over the wall-clock time. There are slightly more remote accesses on threads 4, 5, 6, and 7, and they spend more time on communication.

Next we turn on the mapping to the source lines, and rerun the experiment. The overhead is still within 8% of the execution time. We note that the tracing overhead does not increase with the configuration size (e.g., roughly the same overhead is observed for an execution with a larger input on a 16 node cluster with each node running 16 threads). The reason is that for each remote memory access, compared with the actual communication time (going through the software stack and the physical network) the time spent on logging is miniscule. There are 3 shared arrays declared in the function (whose main loop is shown in Fig. 5) that computes the connected components. Fig. 3 is a pie chart showing the relative number of remote accesses to different shared arrays. Most remote accesses are to D, and there is no remote access to *El*. Fig. 4 illustrates the number of remote accesses to shared array P from each individual thread. We can see they are roughly balanced among the threads.

Profiling clearly suggests that tuning should focus on array D. D is referenced at many lines in the code shown in Fig 5. The *while* loop in the figure is divided into two parts, *grafting* of components from lines 27 to 40 and *shortcutting* from lines 45 to 49. It is unclear yet which of the two segments dominates the remote access time.

We construct the distribution of remote accesses on the source lines as shown by the second column in Fig. 5. We see that most remote accesses happen inside the *grafting* region, specifically, on line 30. Lines 31 and 32 deserve a little discussion. Notice that $D[u]$ and $D[v]$ are already retrieved on line 30. At first glance, the number of remote accesses on lines 31 and 32 should be the same as the distribution of D and P are identical. Yet there are roughly 2 times more remote accesses on line 32. This is due to the fact that array D is updated on line 31, and the compiler generates code that reloads the values of $D[u]$ and $D[v]$.

The remote access time distribution can be similarly constructed, and we omit those results due to limited space.

```
25          while(1) {
26                  grafted = 0;
27                  upc_forall(i=0; i<m; i++; &El[i]) {
28                          u = El[i].u;
29                          v = El[i].v;
30   15027                  if(D[u] < D[v]){
31    2074                          D[D[v]] = D[u];
32    5981                          P[D[v]] = D[u];
33                                  grafted = 1;
34                          }
35                          else if (D[v]<D[u]){
36    2084                          D[D[u]] = D[v];
37    6142                          P[D[v]] = D[v];
38                                  grafted = 1;
39                          }
40                  }
41                  upc_barrier;
42
43                  grafted = all_reduce_i(grafted, UPC_ADD);
44                  if(grafted == 0) break;
45                  upc_forall(i=0; i<n; i++; &D[i]) {
46     1991                   while(D[i] != D[D[i]]) {
47     1313                         D[i] = D[D[i]];
48                          }
49                  }
50                  upc_barrier;
51          }
```

Figure 5. Distribution of remote accesses on the statements on thread 0. Both D and P are integer arrays. *El* is an input edge list. Each edge has two endpoints (u,v). $D[i]$ represents the connected component that vertex i belongs to. All arrays have default distribution. There is no P in the original CC algorithm, and we add P to record the grafting and to showcase the analysis using our tools

4. Improving communication efficiency through caching and coalescing

Section 3.2 shows that CC incurs many remote accesses. Large aggregate network latency resulted from numerous small messages is the root cause of poor performance for many UPC programs. To improve communication efficiency, our tool provides refactorings for optimizing remote memory accesses through software caching and message coalescing. Although in theory both can be implemented by the compiler [4], in practice, due to the implications of memory consistency and lack of runtime feedback, caching and coalescing transformations are provided by very few compilers. In fact, neither the IBM UPC compiler nor the Berkeley UPC compiler [15] optimizes CC with the two transformations. In this section we present our implementation of source code refactorings for caching and coalescing. Our optimizations focus on *upc_forall* regions, and lend themselves to easy customization from the user. Currently our tool does not support automatic refactorings for caching and coalescing outside *upc_forall* regions. We are aware that there exist codes, especially the ones that are translated directly from prior MPI implementations, that do not employ *upc_forall* constructs. In general it is hard to optimize them through only source code refactorings.

4.1 Caching remote accesses

For each communication-intensive *upc_forall* statement, we maintain at each thread a *software cache* to provide fast access to each shared data structure that incurred many remote accesses in the profiling run. The cache is implemented using a hash table with the address to the shared data element as the hash key. Various hash functions may be used, and we find simple linear hashing functions suffice for our purpose. When an access to a remote address hits the cache, the value in the table is returned; otherwise a remote request is issued to bring in the data. Software caching has appeared in earlier studies (e.g., [4]). Our contribution is the design of efficient caching for UPC programs for data intensive applications.

As the iterations in *upc_forall* are parallel, our software cache implements relaxed memory consistency, that is, the completion order of memory accesses can be arbitrary on different threads. Relaxed consistency provides much freedom as to whether and how to update data held at remote caches. In fact most UPC programs adopt relaxed consistency models for better performance. We allow a software cache line to be held at multiple caches in valid state simultaneously regardless of remote updates. Dirty copies are flushed out at the synchronizations points (e.g., *upc_barrier*). When flushing out, different copies of the same cache line race to update the memory, and depending on the policy tunable by the user, one of them wins. For example, if a copy from any arbitrary processor may win, the memory model is similar to concurrent-read-concurrent-write PRAM (CRCW-PRAM); if we define a priority on the copies from different threads, we can simulate priority CRCW-PRAM.

We adopt an extreme cache replacement policy to avoid expensive runtime cache coherence traffic. Our caches can be considered

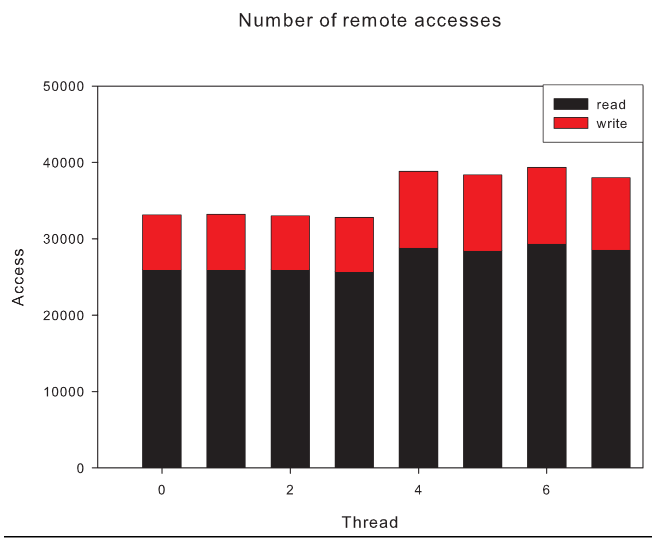

Figure 1. Number of remote accesses

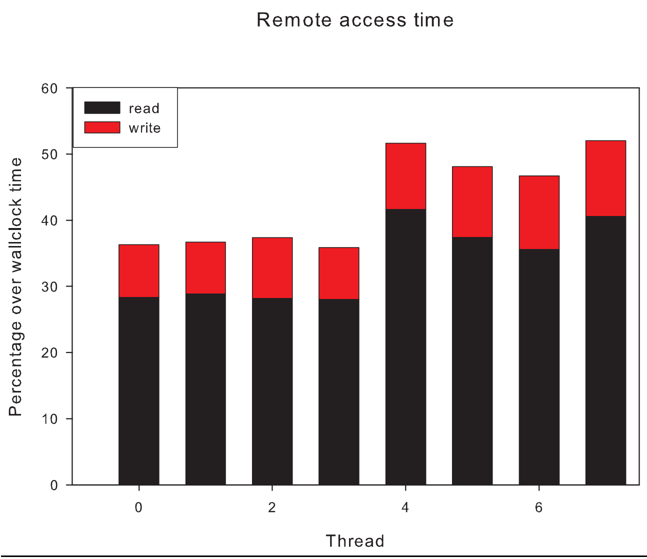

Figure 2. Remote access time

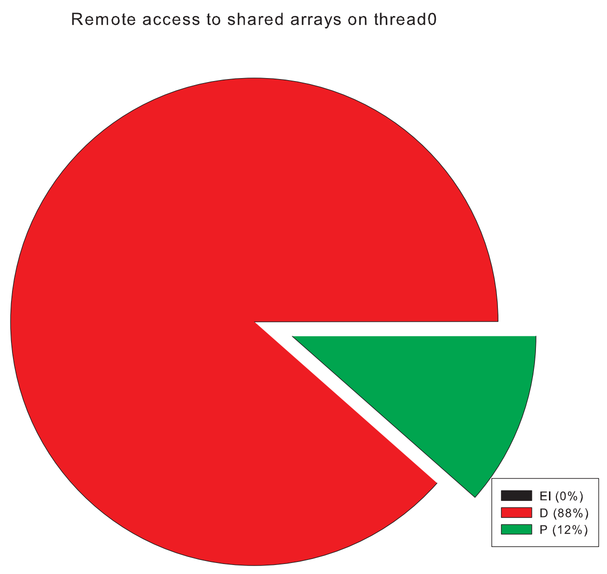

Figure 3. Accesses to shared arrays

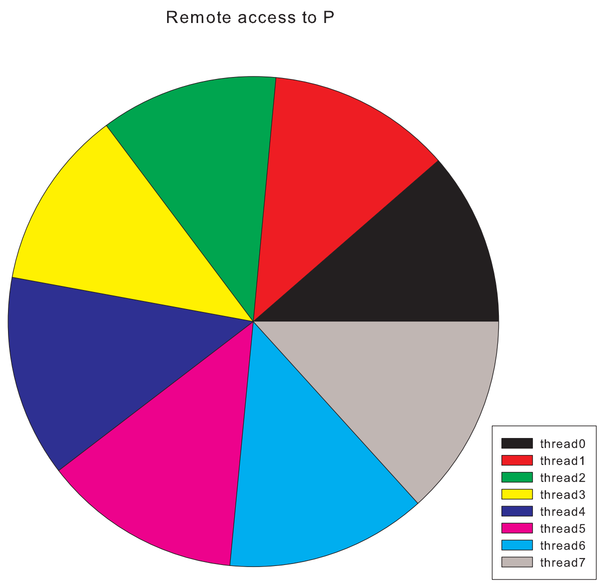

Figure 4. Remote accesses to P

as fully associative with "unlimited" capacity, and there is no cache line replacement at all during the loop execution. Under this policy, the hash table size can grow to several times of the input size. Memory consumption usually is not a big concern for software caches. For data intensive applications, however, there may not be enough extra memory for the cache to grow. To maintain zero cache line replacement inside the loop, we tile the *upc_forall* iterations to limit the memory consumption of caches. Consider the following example

```
    upc_forall(i=0; i<n; i++; i) {...}
```

The software cache size is determined by n. According to the available memory size C, the loop is tiled as follows

```
    for(ii=0; ii*C <n; ii++)
      upc_forall(i=ii*C, i<min((ii+1)*C, n); i++; i) {...}
```

The tiled scheduling effectively reduces the cache size (now proportional to C), and pays the penalty of $\lceil \frac{n}{C} \rceil$ rounds of communication to flush the dirty lines.

Customizing the parameter C and the race arbitration policy are exposed to the user through our automated source code refactoring facility.

4.2 Coalescing

Caching is effective when the program exhibits good temporal locality with frequent data reuse. For parallel programs with poor temporal locality, we combine software caching with message coalescing [4] to further improve communication performance.

For shared array accesses whose indices can be determined without requesting remote data, the indices are computed in advance, and the corresponding values are prefetched into the local caches in batches. This can be implemented using several *all-to-all* or similar communication primitives [3]. As a result, many short messages are coalesced into a few long ones, and the adverse impact of network latency is alleviated. Similarly, writing *dirty* data to remote locations is also implemented with coalescing. The batched reads and writes occur outside the *upc_forall* loop. During the execution of the loop, access to prefetched data becomes local, and various prefetching algorithms can be used to prefetch data that are not already brought in. Again the customization of the prefetching is exposed to user through our refactoring tool.

4.3 Automated refactorings

Caching and coalescing are implemented through source code refactoring with a support library in our tool. The programmer can either use the library manually to add caching support for the access of shared data structures, or she can specify the data structures and have our utility automatically refactor the program. The utility is available under IBM alphaWorks [6], and interested readers are welcome to give it a try.

As we target only *upc_forall* statements, the static analysis involved in the refactoring is straightforward. The iterations may be scheduled in any order. Within an iteration we maintain program order and preserve synchronization semantics among threads at the barriers. We next give an example of refactoring UPC programs.

The following UPC code computes the dot product of two vectors. Vector x is of size n with default distribution, while vector y is of size $2n$ also with default distribution. The computation is $x_i \leftarrow x_i \times y_{2i}$.

```
1: int dp(int n, shared int *x, shared int *y)
2: { int i;
3:   upc_forall (i = 0; i < n; i++; &x[i]) {
4:     x[i] = x[i] * y[i * 2];
5:   }
6:   return 0;
7: }
```

In the code at line 4 access to *x* is local as specified by the *upc_forall* statement (&*x*[*i*] states that the thread owning *x*[*i*] executes the computation). Access to *y*[2*i*] is remote for most i, $i \in [0, n-1]$. Profiling shows that remote accesses are entirely towards *y*, the refactored version with caching, coalescing, and cache size management is shown below.

```
 1: int dp(int n, shared int *x, shared int *y)
 2: {
 3:   int i, chunk, ii, ty;
 4:   chunk = 1024;
 5:   hpcstpf_open(y, chunk);
 6:   for (ii = 0; ii * chunk < n; ii++) {
 7:     upc_forall (i = ii * chunk; i < MIN((ii + 1) * chunk, n);
 8:                 i++; &x[i]) {
 9:       hpcstpf_hint(hdl, i * 2);
10:     }
11:     hpcstpf_start_download(hdl);
12:     hpcstpf_finish_download(hdl);
13:     upc_forall (i = ii * chunk; i < MIN((ii + 1) * chunk, n);
14:                 i++; &x[i]) {
15:       hpcstpf_get(hdl, i * 2, &ty);
16:       x[i] = x[i] * ty;
17:     }
18:     hpcstpf_start_upload(hdl);
19:     hpcstpf_finish_upload(hdl);
20:   }
21:   hpcstpf_close(hdl);
22:   return 0;
23: }
```

In the refactored program, the loop is first tiled with a tile size 1024. *hpcstpf_open* at line 5 is a function from our library that prepares a cache for the shared array *y* with cache size *chunk*. *hpcstpf_hint* at line 9 computes indices to shared array *y* used in the loop. *hpcstpf_start_download* and *hpcstpf_finish_download* at lines 11-12 prefetches data in batches to the cache. *hpcstpf_get* at line 15 accesses the cache and returns the value for the specified array index. *hpcstpf_start_upload* and *hpcstpf_finish_upload* at line 18-19 flushes the dirty copies of y, if any (in this example y is clean), to update the shared array. *hpcstpf_close* at line 21 destroys the software cache and releases allocated resources.

5. Tuning results

We use our tool to assist the tuning of two programs that are either irregular or have a significant amount of communication. They are connected components (CC) and 2D stencil computation (Stencil). CC represents a class of sparse, graph problems with highly irregular memory access pattern. Stencil has the typical access pattern of many scientific computations where the value of an attention point is computed using values from adjacent points. Remote accesses occur at the 2D borders of the data grid on each processor. Profiling shows that in both programs there are many remote accesses to certain shared data structures.

Our target platform is a cluster of IBM P575+ nodes connected with a dual-plane 2GB/s High-performance Switch (HIPS). Each node is configured as 16 CPUs running at 1.9 GHz with 64GB DDR2 memory. This setting is typical of clusters of SMPs that are the basis of many supercomputers. The cluster has 32 nodes, and is shared by many bench-markers. Due to limited availability and the long execution time of naive UPC implementations, we can not run all experiments using all 32 nodes. We provide performance results on 32 nodes only for the more interesting case, that is, the highly irregular CC.

For CC we use an input of random graph with 100 million vertices and 200 million edges. The number of nodes varies from 8 to 32. Fig. 6 compares the performance of two programs. *Original* shows the performance of the original program. *Manual* shows the performance of the manually-tuned program as described in [3]. In [3], the shared-memory accesses in CC are scheduled in a recursive fashion for improved locality behavior that resulted in better communication and cache-performance. Various problem-specific and UPC-specific optimizations (such as contracting the edge list and privatizing local accesses) are also presented in [3]. In this study, only optimizations that can be automated in [3] are implemented by *Manual*.

Tool optimized shows the performance of the program optimized by our tool. The bars in Fig. 6 are divided into 3 groups for the runs with 8 nodes, 16 nodes, and 32 nodes, respectively.

The *Manual* and *Tool optimized* have comparable performance, and they are about 5 times faster than *Original*.

For Stencil we run our experiments with different grid sizes. Fig. 7 shows that using 16 nodes and 2 threads per node, the performance is improved by about 1.7 to 3.5 times with our optimization. When we further increase the number of threads per node, the performance improvement decreases as there are fewer remote accesses on each thread and the "communication" within a node starts to dominate.

6. Conclusion and future work

We presented our study of tool-assisted performance analysis and tuning of UPC applications. In addition to capturing the runtime statistics, our tool is capable of attributing the communication resulted from remote memory accesses to the source. With the insight gained from profiling, we can further use our refactoring tool to implement caching and coalescing for remote accesses. Experimental results show that significant performance improvement is achieved

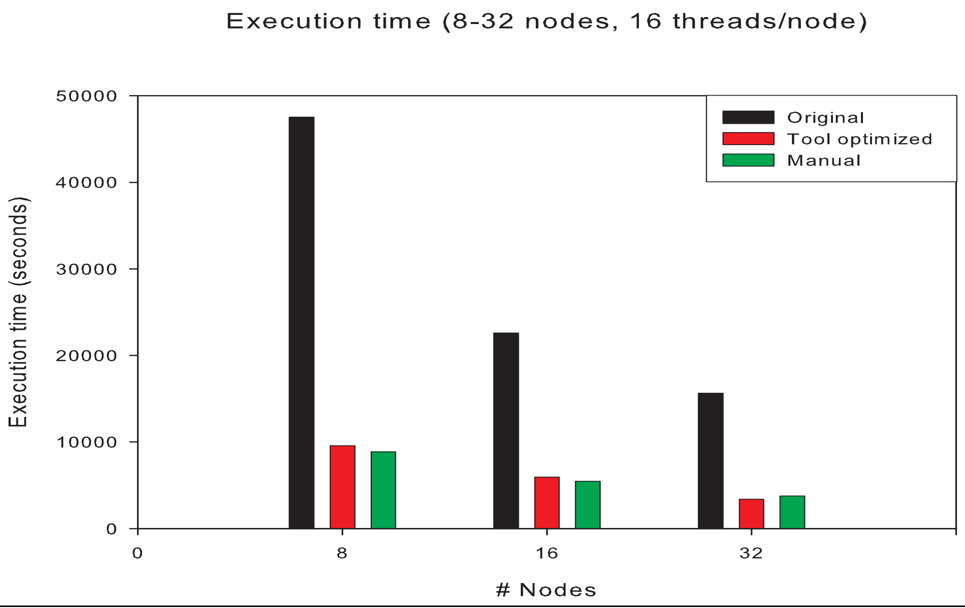

Figure 6. Performance of CC

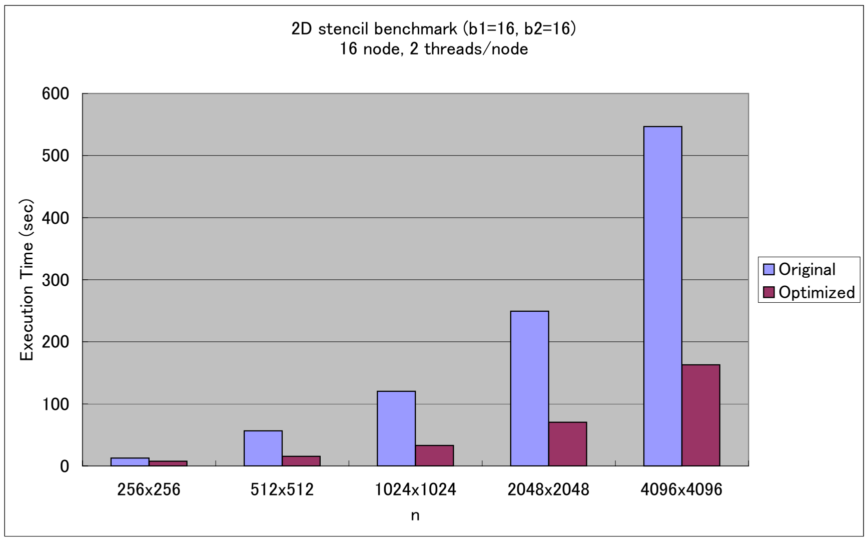

Figure 7. Performance of Stencil

for two UPC programs, each representing a wide range of applications.

In the future we plan to study the shared-memory access patterns including local accesses in UPC applications, and observe patterns for performance problems. We also want to investigate whether a similar strategy can be adapted for other PGAS languages.

References

[1] P. Charles, C. Donawa, K. Ebcioglu, and etc. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 519–538, San Diego, CA, 2005.

[2] W. Chen, D. Bonachea, J. Duell, and etc. A performance analysis of the berkeley upc compiler. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 63–73, New York, NY, USA, 2003. ACM.

[3] G. Cong, G. Almasi, and V. Saraswat. Fast PGAS implementation of distributed graph algorithms. In *Proc. the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC2010)*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[4] R. Das, Y. Hwang, M. Uysal, and etc. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proc. Scalable Parallel Libraries Conference*, pages 45–56. IEEE Computer Society Press, 1993.

[5] M. Hermanns, B. Mohr, and F. Wolf. Event-based measurement and analysis of one-sided communication. In *Proc. Euro-Par 2005*, pages 156–166. Springer, 2005.

[6] High productivity computing systems toolkit. IBM alphaworks. `http://www.alphaworks.ibm.com/tech/hpcst`.

[7] J.L. Martin and J. Dongarra. Special issue on MPI: a message-passing interface standard. *Int. J. Supercomput. Appl. High Perform. Eng.*, 8(3-4), 1994.

[8] B. Mohr, L. De Rose, and J. Vetter. A performance measurement infrastructure for co-array fortran. In *Proc. Europar 2005, Lecture Notes in Computer Science*, volume 3648, pages 146–155. Springer Berlin/Heidelberg, 2005.

[9] S. Sbaraglia, K. Ekanadham, S. Crea, and etc. pSigma: An infrastructure for parallel application performance analysis using symbolic specifications. In *Proc. of the sixth European Workshop on OpenMP*, 2004.

[10] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

[11] H. Su, M. Billingsley, and A.D. George. Parallel performance wizard: A performance system for the analysis of partitioned global-address-space applications. *Int. J. High Perform. Comput. Appl.*, 24:485–510, November 2010.

[12] H. Su, D. Bonachea, A. Leko, and etc. GASP! a standardized performance analysis tool interface for global address space programming models. In *Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06)*, 2006.

[13] *Unified Parallel C*, URL:http://en.wikipedia.org/wiki/Unified␣Parallel␣C.

[14] IBM, *The IBM XL UPC Compiler*, `http://www.alphaworks.ibm.com/tech/upccompiler`.

[15] K. Yelick *et al.*, *The Berkeley UPC Compiler*, `http://upc.lbl.gov/`.